

# 基于影响域分析的软件补丁兼容性检测

(申请清华大学工程硕士学位论文)

培养单位: 软件学院  
学 科: 软件工程  
研 究 生: 潘 晓 梦  
指 导 教 师: 贺 飞 副 教 授

二〇一五年五月



# **Compatibility checking of software patches by impacted analysis**

Thesis Submitted to  
**Tsinghua University**  
in partial fulfillment of the requirement  
for the professional degree of  
**Master of Software Engineering**

by  
**Pan Xiaomeng**  
**( Software Engineering )**

Thesis Supervisor : Associate Professor He Fei

**May, 2015**



# 关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：\_\_\_\_\_

导师签名：\_\_\_\_\_

日 期：\_\_\_\_\_

日 期：\_\_\_\_\_



## 摘要

软件补丁可以在不修改软件自身代码的基础上，对软件的行为进行修补。通过补丁，可以很好的将软件开发过程和软件维护过程解耦合，以较小的代价实现软件系统的缺陷修补和功能增强。然而，软件本身也会自演化。为每个软件版本开发相应的补丁将耗费极大的人力物力。这里的关键问题是检查针对已有的补丁针对新版本的软件是否仍然适用。

本文从语法和语义两个层面分析补丁和不同版本软件之间的兼容性问题，采用静态分析的方法，并引入变更影响域分析技术，能够有效识别与不同版本兼容的软件补丁，避免了重复的补丁开发工作。本文完成的主要工作归纳如下：

1. 补丁兼容性问题归结为不同变更之间的冲突问题。注意到软件补丁和软件演化都是针对软件原始版本的变更，所以补丁兼容性问题的实质就是两次变更之间是否存在冲突的问题。
2. 设计并实现了一套基于影响域分析的补丁兼容性检测方法。主要包含两个步骤。第一步是变更影响域分析，它可以分析不同代码版本间的变更集合，并挖掘这些变更对于软件代码的语义影响，也就是所谓的变更影响域。第二步是软件变更冲突检测，该过程根据上一步中得到的变更影响域，完成变更间的语义冲突检测。
3. 在 Eclipse JDT Core 项目上进行了实验。实验结果表明了该工具的可用性和有效性。

**关键词：**软件维护；变更影响域；补丁；兼容性检测；软件演进

## Abstract

Patch is capable of fixing software behavior without editing the source code. By using patch the process of software development and software maintenance could be uncoupled so that the cost of bug fixing and functionality augmenting process can be reduced. However, software itself will evolve and developing corresponding patch for each version of the software is too costly. The core issue here is how to determine whether the patch is applicable for other software versions.

This thesis analyzes the compatibility problem between patch and different software versions viewed from both syntactic and semantic perspectives and propose a static analysis method combined with impacted analysis, making the process of recognizing the patch which is compatible with another software version effective and avoiding replicate patch developing work. The major work of this thesis are as follows.

1. The compatibility checking process is actually the conflict checking process between changes. Considering that the patch and software evolving process are both changes on the original software version, the compatibility problem is essentially the problem of determining whether the two changes are conflicted.
2. Design and accomplish a patch compatibility checking method by impacted analysis. The method contains two sub-analyses. At first we use a so called software change impacted area analysis to get the change set between different versions of codes and try to find out its semantic impact on other syntactic structures, namely change impacted area. Then we use the change impacted area to accomplish the software change conflict checking process, this process will check whether there exists semantic conflict between the changes.
3. After experimenting on the Eclipse JDT Core project, the tool is proven to have the ability to find out compatibility issues between the patches so that this solution is applicable and sound.

**Key words:** software maintenance; change impacted area; patch; compatibility checking; software evolving

## 目 录

|                                |           |
|--------------------------------|-----------|
| <b>第 1 章 绪论 .....</b>          | <b>1</b>  |
| 1.1 研究背景 .....                 | 1         |
| 1.2 研究内容 .....                 | 2         |
| 1.3 本文组织结构.....                | 3         |
| <b>第 2 章 相关工作 .....</b>        | <b>4</b>  |
| 2.1 程序间差异性分析 .....             | 4         |
| 2.2 程序变更影响分析 .....             | 5         |
| 2.3 相关工具 .....                 | 7         |
| 2.3.1 git.....                 | 7         |
| 2.3.2 Beyond Compare .....     | 7         |
| 2.3.3 jpf-regression .....     | 7         |
| 2.3.4 ASTro .....              | 9         |
| <b>第 3 章 软件补丁兼容性检测方法 .....</b> | <b>10</b> |
| 3.1 兼容性问题.....                 | 10        |
| 3.2 检测方法概述.....                | 11        |
| 3.3 本章小结 .....                 | 13        |
| <b>第 4 章 软件变更影响域分析 .....</b>   | <b>14</b> |
| 4.1 程序间语法差异性分析.....            | 15        |
| 4.1.1 相关定义.....                | 15        |
| 4.1.2 分析方法.....                | 15        |
| 4.1.3 模块设计与实现 .....            | 16        |
| 4.2 变更语义影响分析 .....             | 20        |
| 4.2.1 相关定义.....                | 20        |
| 4.2.2 分析方法.....                | 21        |
| 4.2.3 模块设计与实现 .....            | 22        |
| 4.3 本章小结 .....                 | 29        |

## 目 录

---

|                                    |           |
|------------------------------------|-----------|
| <b>第 5 章 软件变更冲突检测方法 .....</b>      | <b>30</b> |
| 5.1 相关定义 .....                     | 30        |
| 5.2 分析方法 .....                     | 30        |
| 5.3 模块设计与实现 .....                  | 31        |
| 5.4 本章小结 .....                     | 34        |
| <b>第 6 章 实验结果与分析 .....</b>         | <b>35</b> |
| 6.1 实验设计 .....                     | 35        |
| 6.2 实验案例 .....                     | 36        |
| 6.3 实验结果与分析 .....                  | 37        |
| 6.3.1 版本迁移 .....                   | 37        |
| 6.3.2 影响域分析模块 .....                | 40        |
| 6.3.2.1 差异性分析模块 .....              | 40        |
| 6.3.2.2 影响分析模块 .....               | 42        |
| 6.3.3 冲突判定模块 .....                 | 43        |
| 6.4 本章小结 .....                     | 46        |
| <b>第 7 章 结论 .....</b>              | <b>47</b> |
| 7.1 工作总结 .....                     | 47        |
| 7.2 未来工作 .....                     | 48        |
| <b>参考文献 .....</b>                  | <b>49</b> |
| <b>致 谢 .....</b>                   | <b>51</b> |
| <b>声 明 .....</b>                   | <b>52</b> |
| <b>个人简历、在学期间发表的学术论文与研究成果 .....</b> | <b>53</b> |

## 第1章 绪论

### 1.1 研究背景

软件维护（Software Maintenance）是软件开发周期中耗时最长、开销最大的过程<sup>[1]</sup>。随着外部环境和用户需求的不断变化，软件系统需要随之进行适应和调整，并修复在实际运行中暴露出来的问题。

软件演进是一类维护过程中常见的活动<sup>[2]</sup>。在演进过程中，软件系统可能会由于各种不同的原因发生更新行为。为了实现该过程，现代软件工程提出了许多解决方案，而补丁（Patch）就是其中一类可用于完成程序漏洞修复、软件功能增强、程序性能改善等任务的方案。补丁一般是 Diff 工具产生的文本数据<sup>[4]</sup>，用于表示以行为单位的程序间文本差异性。它在工业界中得到了广泛应用，是软件维护过程的重要组成部分<sup>[3]</sup>。

然而，在实际应用中补丁程序仍具有一定的局限性。由于补丁一般只针对软件的某个专门版本开发，对软件演进过程中不断推出的新版本而言，无法确定补丁程序是否同样适用。再加上补丁程序一般都具有特定的应用目的，例如功能升级、漏洞修补等，使得新版本可能仍然需要应用原有补丁来完善自身，因此在实际的软件维护过程中工程师往往不得不重新开发针对该新版本的特定补丁程序，对人力和时间等资源造成了许多浪费。

可以考虑这样一个应用场景：

- 某项目团队在开发过程中使用了某开源第三方软件，并针对该开源软件开发了专门的补丁以适用于本项目。
- 当第三方软件更新到新版本，如果集成该新版本，原有的补丁是否还适用？

关键在于，补丁是否能够在不同的软件版本之间进行共享，即如何确定补丁对于其他软件版本的兼容性。究其缘由，补丁兼容性问题主要在于补丁程序会引入软件变更（Software Change）<sup>[5]</sup>，应用这些变更可以实现功能修复、版本升级等目的。然而，软件变更的影响不仅仅局限于被修改的某行，由于软件代码的耦合性，单行变更就足以将其影响传播到软件系统的其他部分<sup>[6,7]</sup>。因此，软件变更不仅会使软件系统发生语法结构上的变化，还会进一步引入语义上的变化。例如，对赋值语句的修改可能会影响到在其后引用该变量的条件语句。

由此可见，如何确定补丁程序对于其他软件版本的兼容性是一个较复杂的问题。根据上文的叙述可知，该问题是由于补丁程序所引入的软件变更和版本升级所引入的软件变更可能发生冲突而造成的，因此该问题的核心在于如何找到变更

间的冲突，这种冲突既可能是语法结构上的，也可能是语义行为上的。

对于语法冲突而言，这主要是由于应用过程中变更破坏了代码的语法结构。这种语法结构上的错误可能是多种多样的，例如补丁所要修改的代码不在原位置或已被删除、补丁所要添加的代码已经在该文件中存在等等。对于语义冲突而言，这主要是由于变更对于某行修改可能会影响到代码中的其他语法结构，从而导致程序的行为发生变化，而这种语义变化是不易察觉的。

综上所述，语法冲突好发现、易解决，现代的版本控制系统等工具都能够检测并修复之。而语义冲突的检测较为复杂，目前学术界尚无这方面的相关工作。因此，本文将主要考虑对软件变更间的语义冲突进行检测。更详细的问题讨论可以参考章节3.1。

## 1.2 研究内容

为了找到补丁与新版本代码间的语义冲突，本文提出了一种基于影响域分析的软件补丁兼容性检测方法，该方法可以找到软件代码在不同版本间引入的变更，并根据这些变更去找到受其语义影响的其他程序代码结构集合，也就是所谓的语义影响域。

在找到这些变更的语义影响域（下文简称“变更影响域”）之后，就可以进行冲突检测。通过分析这些变更影响域之间是否存在重叠，就能够发现补丁对旧版本造成的语义变化是否会影响到旧版本在演进时所引入的语义变化，也就是说，代码在重叠的位置可能存在冲突。目前而言，冲突的确定需要进一步的人工分析来完成。

在软件补丁兼容性检测问题上，本文的主要贡献包括以下几个部分。

首先，本文对软件补丁兼容性检测问题进行了分析。补丁的兼容性问题主要在于补丁的变更所造成的语义影响之间存在冲突，如何找到这样的语义冲突是解决该问题的重点。

其次，为了找到这样的语义冲突，本文提出了一套补丁兼容性检测方法，该方法能够分析软件版本间的变更，并找到其变更影响域，通过判断变更影响域间是否存在重叠，就能够找到可能存在语义冲突的代码位置。该套检测方法主要包括：

- 软件变更影响域分析：该分析过程通过寻找不同软件版本间的变更集合来分析并找到软件变更对代码的语义影响域，即变更影响域。
- 软件变更冲突检测：该分析过程根据得到的不同变更影响域，分析其是否存在语义冲突。

最后，本文根据该检测方法给出了具体的工具实现，并在工业界的实际项目

上进行了实验。根据兼容性检测工具在 Eclipse JDT Core 项目上的实验结果，本文中所给出的兼容性检测工具是可用的，其检测结果是正确的。

### 1.3 本文组织结构

本文共七章。第一章是绪论，介绍本文的研究背景和主要工作；第二章主要介绍与本文所述内容相关的国内外的工作；第三章主要介绍补丁兼容性检测所要解决的问题、检测方法及其工具实现；第四章主要介绍了检测方法中的软件变更影响域分析方法及其对应的工具模块实现；第五章介绍了检测方法中的软件变更冲突检测过程及其对应的工具模块实现；第六章介绍了实验过程和结果；第七章主要对本文的工作进行了总结，并提出了进一步的工作方向。

## 第 2 章 相关工作

本章中主要介绍与本文相关的国内外工作，主要包括对变更影响域分析的子过程和相关工具的调研。

### 2.1 程序间差异性分析

对于软件演进分析而言，如何确定一个程序的不同版本之间的变更是一个关键性的问题<sup>[8]</sup>。程序间差异性分析能够通过分析同一程序的不同版本间的差异，来确定版本间的变更集合<sup>[9,10]</sup>。

按分析的深度而言，程序间差异性分析可以分为三类：

- 文本差异：单纯对比文本间的不同，这是最简单也最广泛应用的分析方法，如 Unix Diff 工具。
- 语法差异：对比并获得源代码间语法结构上的不同。
- 语义差异：对比并获得源代码间语义层面上的不同。

现有的帮助工程师进行软件维护和演进过程的工具往往都受限于低质量的变更信息。例如，源代码的变更信息往往都存储于版本管理系统中（如 CVS）等，它们会追踪对某个特定文件的文本行的增加/删除等操作，但没有考虑代码中的结构化变更。

考虑到源代码能够以抽象语法树（Abstract Syntax Tree）的形式进行表达，可以采用树间差异分析的方法来抽取出这些变更信息。Change Distilling 就是这样一类进行树间差异分析的算法<sup>[11,12]</sup>。该算法能够从两棵 AST 之间寻找匹配节点，并找到一个能够令一棵树转化为另一棵树的最小变更集合，该变更集合即为所求的程序间差异。而且由于是从 AST 中抽取信息，该算法可以获取语法结构上的变更信息。

Change Distilling 中采用二元字符串相似性来匹配源代码语句，并使用子树相似性来匹配源代码结构（如语句、循环等）。在寻找变更集合时，它采用基本的树变更操作来描述源代码的变更，包括更新、删除、增加等。在实际的使用过程中，该算法可能会受限于如何找到数量合适的移动操作。

## 2.2 程序变更影响分析

软件维护是软件开发周期中最为复杂、成本最高的劳动密集型活动。软件产品需要跟随用户需求的变更而进行适应和变化，而软件变更可以帮助软件实现这种维护过程。事实上，软件变更是软件维护过程中的基础组件，它可能来自于新的需求、缺陷修复、变更请求等，然而将变更应用于软件时，它们会不可避免的带来一些副作用，导致可能会与原软件的其他部分发生冲突。

而变更影响分析（Change Impact Analysis）正是这样一类用于确定变更对于软件其他部分影响的技术集合<sup>[13]</sup>，它在软件开发、维护和测试等过程中都起到重要的作用<sup>[14]</sup>。一般而言，变更影响分析可以用于程序理解、变更影响预测、影响追踪、变更传播、测试用例的选取等过程。

变更影响分析方法可以分类如下：

- 基于可追踪性的变更影响分析<sup>[15]</sup>，它追踪两个不同抽象级别的软件元素之间的依赖性，其目的在于链接不同类型的软件工件（如需求、设计等）。
- 基于依赖的变更影响分析<sup>[16]</sup>，它致力于衡量变更的潜在影响，并试图分析程序语法结构间的关系，即程序实体间的语义依赖。这类变更影响分析主要在源代码级别进行研究。

软件变更可能导致意料之外的副作用，而变更影响分析的目的就在于找到这些副作用（Side Effect 或者 Ripple Effect）<sup>[17]</sup>，并防止之。变更影响分析从分析变更请求和源代码开始，最后能够得到估计影响集合（Estimated Impact Set），与真实影响集合（Actual Impact Set）相比该结果可能存在一定的误差。

整个软件变更影响分析的过程可以大致划分为如下流程<sup>[15,18]</sup>，更详细的流程可以参考图2.1。

该分析过程需要输入变更集合，然后对变更请求进行分析。该步骤即特征定位（feature location），用于找到源代码中相应功能的起始实现位置<sup>[19]</sup>。该分析过程将衡量变更集合引入的影响。该步骤是目前大部分变更影响分析技术的重点，其主要的两类包括：

- 静态分析：包括历史分析、文本分析、结构分析等<sup>[20,21]</sup>。静态分析主要分析程序的语法、语义或者历史依赖，容易产生许多误报（False Positive）。
  - 结构分析着重于分析程序间的结构依赖性并构建依赖关系图
  - 文本分析根据程序中的注释和标识符提取出其概念依赖性
  - 历史分析能够从多个软件版本的演进过程中挖掘相关信息
- 动态分析：包括在线分析和离线分析。该分析过程需要给出特定输入，并依赖程序运行时所收集到的信息来进行分析（如运行时的路径追踪和覆盖信息）

等)[22]。该分析过程所得到的影响集合往往比静态分析的精度更高，但其开销也相应更大，且容易错报（False Negative）。

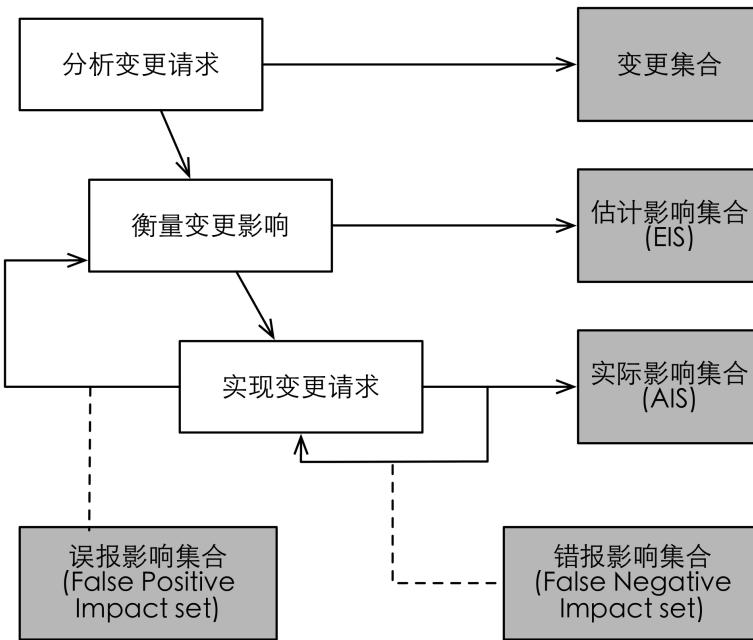


图 2.1 变更影响分析过程

近年来，学术界中实现了某些以变更影响分析技术为支撑的工具，这些工具通常会利用变更影响分析得到的影响集合来完成后续的分析过程，帮助软件进行维护和演进。下面给出相关的简要介绍：

- Chianti：支持 Java 语言，可作为 Eclipse 插件使用<sup>[23]</sup>。该工具首先使用回归测试来分析变更前的程序是否能正常使用，若回归测试失败，则利用 Chianti 工具实施变更影响分析，该分析过程通过将软件变更拆分成若干原子变更来分析变更之间的依赖关系。最后结合原程序生成某种中间表示形式并找到可能影响到测试用例运行的代码位置。
- JRipples：支持 Java 语言，可作为 Eclipse 插件使用<sup>[24,25]</sup>。该工具利用依赖关系图自动标注可能被变更的类所影响的其他类，并提示用户其变更的可能影响传播路径。该工具的分析结果可进行人工修正。
- ROSE：支持用 CVS 工具进行版本管理的 Java 项目，可作为 Eclipse 插件使用<sup>[26]</sup>。该工具需要挖掘软件代码仓库，当用户对代码进行变更时，提示用户某些其他变更可能与之相关（其形式类似于“变更了该函数的人通常还变更了另一个函数”）。
- jpf-regression：支持 Java 语言，可用作 Eclipse 插件或直接作为命令行工具使用<sup>[27]</sup>。该工具利用程序切片技术进行变更影响分析，使用得到的影响集合

来驱动符号化执行，找到可能被变更影响到的程序行为。

## 2.3 相关工具

本节主要介绍实验采用的相关工具。

### 2.3.1 git

git 是一个分布式的版本控制系统，最初由 Linus Torvalds 在 2005 年为 Linux 内核而开发，现在已经成为最流行的版本控制系统。

与 CVS 和 SVN 等集中式的 C/S 版本控制系统不同，git 是一种分布式的版本管理系统，每个本地 git 工作目录都具有完整的历史数据和版本追踪能力，无需网络连接或服务器端的支持。

本文主要采用 git 作为版本管理与合并的工具。

### 2.3.2 Beyond Compare

Beyond Compare 是一款内容比较工具，可用于文件、目录、压缩包等数据间的比较，横跨 Windows、Mac OS X 和 Linux 三大操作系统，可用作常见版本控制系统的第三方文本比较与合并工具。

本文主要采用该工具作为文本比较和合并工具，用于解决补丁版本迁移时可能遇到的冲突。

### 2.3.3 jpf-regression

变更影响分析常用于衡量软件变更的潜在影响，其分析结果通常可用做其他程序分析技术的输入，例如回归测试可利用变更影响分析来确定程序的哪些部分需要进行再分析。由于单行变更就足以引发广泛的未知影响，变更影响分析在软件的演进和维护过程中扮演着重要的角色。<sup>[28]</sup>

目前，大部分的自动分析工具都以程序语法结构的形式描述变更的影响，如函数和语句等。基于依赖的分析方法一般通过分析程序组件间的内部关系来衡量变更的影响，这类技术通常使用程序位置信息来描述变更的影响，缺失了受影响代码位置的相关运行路径信息。这类信息往往对程序行为的验证、调试等工作帮助很大，而且能够将需要关注的代码范围缩小，使得只需关注受影响的程序行为集合即可。

(Directed Incremental Symbolic Execution) DiSE<sup>[27,29]</sup> 方法能够结合静态分析的效率和符号化执行的精度等优点，该方法能分析限制在方法内部的变更影响，并描述程序变更对其行为的影响。

jpf-regression 是 DiSE 方法的工具实现，它基于 Java Path Finder 框架<sup>[30]</sup> 实现，支持 Java 语言，可作为 Eclipse 的插件使用。本文将采用该工具来实现变更语义影响分析过程。下面对该方法中的影响分析算法进行简介。

DiSE 方法采用程序切片技术来衡量变更对代码中其他部分的影响，其生成的影响集合可以用于引导符号化执行来分析受变更影响的程序行为，并生成相应的路径条件（Path Condition），这些路径条件描述了受影响的程序行为，在分析结束后可以利用 SMT 等技术进行路径求解，用于后续的验证、调试等过程。

在 DiSE 方法中，程序变更影响分析是其后续分析过程的基础，其变更影响分析技术的主要特点包括：

1. 粒度：基本块。即变更影响分析过程中，以基本块为单位进行影响集合的计算。
2. 影响范围：方法内部。即将单次影响集合并计算的范围局限于方法内部，最后得到变更对其所属方法内部的其他语句所造成的影响。
3. 影响来源：主要从控制流和数据流两个方面考虑变更所造成的影响，采用语句间的控制依赖和数据依赖关系进行影响计算。

DiSE 方法中的影响集合主要分为两类：

- ACN：受影响的条件语句节点（Affected Conditional Nodes）。
- AWN：受影响的赋值语句节点（Affected Write Nodes）。

为了得到这两类影响集合，DiSE 中使用四条规则进行迭代计算。从原始的变更集合出发，不断应用规则向外扩展，最后得到的闭包即为所求的影响集合：

1. 如果 ACN 中有一个节点  $n_i$ ，且存在一个条件节点  $n_j$ ，其中  $n_j$  控制依赖于  $n_i$ ，那么将  $n_j$  加入到 ACN 中。
2. 如果  $n_j$  是一条赋值语句，且控制依赖于 ACN 中的节点  $n_i$ ，那么  $n_j$  加入到 AWN 中。
3. 如果 AWN 中的一条赋值语句节点  $n_i$  对于变量的赋值在条件语句  $n_j$  中被使用了，且 CFG 中存在一条从  $n_i$  到  $n_j$  的路径，那么将  $n_j$  加入到 ACN 中。
4. 如果一个写语句节点  $n_i$  对于变量的赋值在 AWN 或 ACN 中的某个节点  $n_j$  被使用了，且 CFG 中存在一条从  $n_i$  到  $n_j$  的路径，那么将  $n_i$  加入到 AWN 中。

### 2.3.4 ASTro

本文采用的程序间语法差异性分析工具是由内布拉斯加大学林肯分校的 Josh Reed, Suzette Person 和 Sebastian Elbaum 等人所开发的 ASTro，它是 jpf-regression 工具自带的前置工具，用于比对两个不同版本的源代码并获取其抽象语法树上的差异，并将得到的变更集合以 XML 格式输出。该工具支持 Java 语言。

## 第3章 软件补丁兼容性检测方法

本章主要对兼容性问题和其检测方法进行概括介绍。

### 3.1 兼容性问题

如前所述，随着软件版本的演进，往往会发生专为某版本设计的补丁无法适用于新版本代码的情况，而如果要为新版本代码重新开发适用的补丁，又耗时耗力极大。因此，本文研究在软件演进背景下，补丁对于其他软件版本的兼容性问题。该问题的主要难点在于如何检测补丁应用于其他软件版本的代码时是否发生冲突。

回顾章节 1.1 中提到的应用场景，某开发团队在其项目中使用了某开源第三方软件，并针对该项目的实际需要对其开发了专门的补丁。现在第三方软件进行了版本升级，该团队需要知道，如果在项目中集成该新版本，原补丁是否还能适用。

考虑到版本更新也可以使用补丁来完成，该问题就可以从另一种角度来看待，即在集成该新版本的时候，检测原补丁是否能够继续使用的过程可以转化为检测原补丁是否会和升级补丁产生冲突的过程。也就是说，软件补丁对于其他版本的适用性可以从补丁之间的冲突这一角度来考虑。

由于每个补丁都可以视作一系列的变更集合，其中的每条变更都会修改原有代码版本的语法结构，并可能会对其他语法结构造成语义上的影响，那么从这一角度来看时，补丁的兼容性检测问题就得到了简化。如果能够找到每个补丁中的变更所影响到的程序语法结构的集合，那么通过判断这两个集合之间是否存在一定的交集就能够确定补丁间的语义影响是否会相互覆盖。

显然，如果两个补丁中的变更影响到了相同的语法结构，由于该位置上的语法结构受到了双重影响，该位置就可能出现冲突。这是由于该位置上的语法结构受到了双重影响，而这些影响之间可能存在冲突，无法共存。例如对某条件判断语句而言，原补丁在某处对其引用的变量值进行了修改，使得条件语句中该值增加，而升级补丁中在另外一处对该变量的修改则会导致条件语句中其值减少，显然，这样的双重影响是矛盾的。

当然，在某些情况下，这样的双重影响也是可以共存的。例如上文条件判断语句的例子中，如果两个补丁中的修改都会导致该条件语句中引用的某变量值增加，那么这样的双重影响就可能是兼容的。

这类双重影响并没有严格的规则来判断是否一定冲突或不冲突，因此，只能通过人工分析来辅助判断。

综上所述，软件补丁的兼容性检测问题可以归结为多次变更之间的冲突检测问题。

## 3.2 检测方法概述

根据上文的讨论，本文提出了一种软件补丁兼容性的检测方法，该方法的整体流程可以参考图 3.1。该方法分为三步。首先，将软件补丁应用于其他版本的代码上，并防止该过程中引入语法错误。其次，找到补丁中的变更所影响到的其他语法结构，也就是后文中提到的变更影响域。最后，根据找到的变更影响域进行冲突检测。该方法的输入包括  $v_1$  版本和  $v_2$  版本的代码，以及适用于  $v_1$  版本的补丁  $p$ 。其中版本  $v_1$  为旧版本，版本  $v_2$  为新版本。

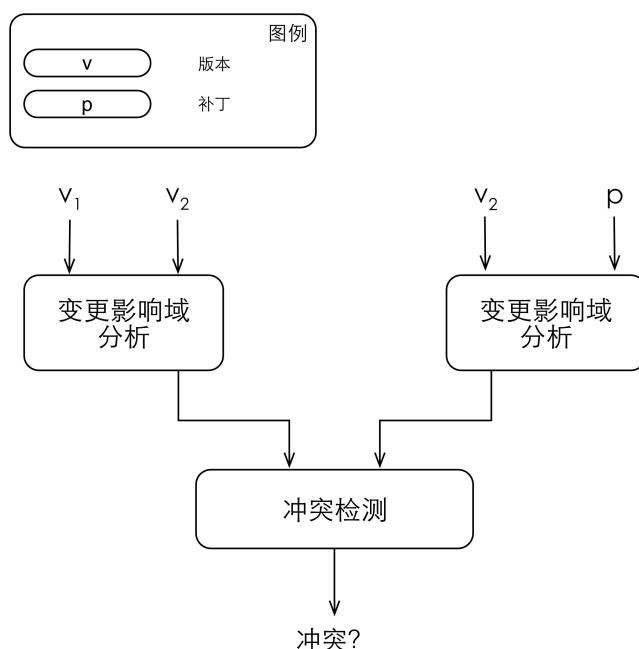


图 3.1 检测方法

该方法可以检测版本  $v_1$  到版本  $v_2$  的升级过程中所引入的变更是否会与补丁  $p$  在版本  $v_2$  中所引入的变更发生冲突。之所以选择以版本  $v_2$  作为基准来进行冲突检测，是由于在实际过程中，应用该检测方法之前还需要将补丁  $p$  应用至软件版本  $v_2$ ，以完成补丁应用的过程。

本文采用了版本合并的方法来解决该过程中可能引入的语法错误，其流程简述如下：

1. 将补丁  $p$  应用到版本  $v_1$ , 获得旧版本应用补丁后的代码, 其版本为  $v_3$ 。
2. 采用三路归并算法将版本  $v_2$  和  $v_3$  进行合并, 获得新版本应用补丁后的代码, 其版本为  $v_4$ 。
3. 解决版本合并中可能出现的冲突问题。

最后所得到的  $v_4$  版本代码即为所需的在版本  $v_2$  上应用了补丁  $p$  中变更的新版本代码。

由于该合并过程较为简单, 可以直接使用 git 等版本控制系统完成, 以后的章节中将不再赘述, 只在实验部分给出该过程相应的结果与分析。本文在以后的章节中将直接考虑对从版本  $v_1$  到  $v_2$  的升级过程中引入的变更与从版本  $v_2$  到  $v_4$  的升级过程中引入的变更进行冲突检测。

下面对该检测方法中所涉及到的过程进行概述。

首先, 该检测方法中提出了一种软件变更影响域分析方法, 该分析方法能够找到不同软件版本间的变更集合, 并获取该变更集合所对应的语义影响域, 即变更影响域, 该变更影响域中包含了所有受到变更集影响的程序语法结构。其中变更影响域分析过程可以分为两个子分析过程, 先利用某种程序间语法差异性分析算法对输入的不同版本代码做差异性分析, 得到以语法结构形式描述的变更集合, 再使用某种变更语义影响分析算法去找到这些变更集合对软件代码的变更影响域。变更影响域分析的两个子过程可以自由替换为符合要求的相应算法实现, 以提高检测方法的实用性。更具体的变更影响域分析过程和变更影响域等相关概念的定义可以参考章节4.2中的叙述。

其次, 该检测方法根据得到的不同变更影响域, 使用软件变更冲突检测方法判定这些变更影响域之间是否存在语义上的冲突。该检测方法主要实现了一种较为简单的自动冲突分析算法, 通过计算变更影响域间的重叠来找到可能出现语义冲突的代码位置。更具体的冲突判定则有待进一步的人工分析来完成。该软件变更冲突检测方法的相关实现和定义可以参考章节5.2中的叙述。

最后得到的冲突检测结果即为所求。

本文根据此处提出的兼容性检测方法进行了相应的工具实现。按照兼容性检测方法的流程, 该检测工具可以划分为两个模块, 即影响域分析模块和冲突判定模块。

影响域分析模块主要实现了检测方法中的软件变更影响域分析方法。由于该方法由两个子分析过程组成, 该模块在实现过程中同样划分为两个对应的子模块, 即差异性分析子模块和影响分析子模块。由于这两个子分析过程已有相关的成熟算法实现, 因此该模块采用了相关的分析工具来实现对应的算法, 并按照本文的实际进行了一定的改进, 因此该模块在实现过程中主要关注分析工具的整合和修

改过程。

冲突判定模块则实现了检测方法中的软件变更冲突检测方法。该模块对软件变更冲突检测方法中提出的冲突检测算法进行了相应的实现，通过影响域分析模块给出的变更影响域来判断可能发生冲突的位置，即变更影响域的重叠位置。最后的冲突判定则由人工分析来完成。

以上模块的具体实现过程可以分别参考相关章节中的叙述。

### 3.3 本章小结

本章概括介绍了软件补丁兼容性问题和其检测方法。章节3.1中介绍了补丁兼容性问题。章节3.2中对补丁兼容性检测方法和其工具实现进行了简要介绍。详细的介绍参见后续章节。

## 第4章 软件变更影响域分析

本章主要介绍软件变更影响域分析的相关概念、分析方法和工具模块的设计与实现。变更影响域分析主要对补丁引入的变更进行分析并找到其对应的语义影响域，即所谓的变更影响域，以用于后续的冲突检测。

由于软件变更是对程序语法结构所作出的修改，它会将原有的语法结构变为新的语法结构。因此，考虑到修改前的语法结构与软件中其他语法结构的耦合性，应用软件变更后，这些相关语法结构的行为会受到该软件变更的影响。本文将这些受到补丁中变更影响的语法结构集合称之为变更的语义影响域（即变更影响域）。更详细的定义可以参考章节4.2.1。

找到变更的语义影响域需要从代码中挖掘两类信息：

1. 代码的变更集合，即两个软件版本之间的语法结构差异性。该集合描述了不同软件版本间的语法结构变更情况。
2. 变更的影响集合，即受变更集合语义影响的其他程序语法结构。该集合描述了变更所造成的语义影响范围。

上述两类集合可用于确定软件变更对代码的语义影响域。因此，该分析过程可以分为两个步骤。首先，变更影响域分析需要找到不同版本代码之间的软件变更集合。其次，变更影响域分析应根据找到的软件变更集合，分析并得到其变更影响域。可见，变更影响域分析可以拆分为两个相应的子过程，即找到软件变更集合的子过程和获取其变更影响域的子过程。这两个子分析过程可以分别采用程序间语法差异性分析和变更语义影响分析来实现。

因此，该分析方法所对应的影响域分析模块在实现的过程中也应拆分为两个子模块，即差异性分析子模块和影响分析子模块。这两个子模块将分别实现程序间语法差异性分析和变更语义影响分析的过程。由于学术界中已有许多相关的分析算法实现，因此影响域分析模块在实现中将直接将采用已有的分析工具。本文中工具实现的主要精力将放在子模块的整合和调整过程上。

综上所述，软件变更影响域分析的过程也就是找到软件变更，并分析得到变更影响域的过程。这些找到的变更影响域将作为软件变更冲突检测过程中的输入，并用于判断变更影响域之间是否会产生冲突。

下面将分别对这两个子分析过程和其模块设计与实现进行介绍。

## 4.1 程序间语法差异性分析

如前所述，程序间语法差异性分析主要是寻找不同版本间的软件变更的过程，该过程需要分析代码间的语法结构差异性，并得到相应的变更集合。该过程主要关注程序间在语法结构上的差异性，最后得到的变更集合也将以语法结构的形式进行描述。

近年来学术界有不少程序间语法差异性分析方面的相关工作，并给出了一些较为成熟的工具实现，因此本文考虑采用已有工具来实现其所对应的差异性分析子模块。

### 4.1.1 相关定义

本节主要介绍程序间语法差异性分析过程中所涉及到的相关概念和定义，以更清楚的了解该分析过程的实质。

假设  $C$  是某个版本的源代码文件中按照该语言的合法语法结构组织起来的代码（如抽象语法树的形式），即由该语言的相应  $\mathcal{S}$  组成的集合。 $\mathcal{S}$  是某种语言的合法语法结构，例如 Java 语言中的语句、方法定义、类定义等，由于随不同语言的实际情况而变化，在这里不做更具体的定义。 $v_k$  表示第  $k$  个版本的代码  $C$ ，其中  $k \in \mathbb{N}$ 。

**定义 4.1:**  $\mathcal{P} : \mathcal{S} \times \mathcal{S}$ 。 $\mathcal{P}$  是补丁，也就是变更集合，即一个由  $\mathcal{S}$  的二元组构成的集合。 $\forall c \in \mathcal{P}$ ，有  $c = (s_i, s_j)$ ，其中  $s_i, s_j \in \mathcal{S}, i, j \in \mathbb{N}$ ， $s_i$  和  $s_j$  分别表示变更前和变更后的代码语法结构。

**定义 4.2:**  $diff : C \times C \mapsto \mathcal{P}$ 。该函数用于求解两个不同版本的代码  $v_i$  和  $v_j$  之间的语法差异性，其结果即为变更集合  $\mathcal{P}$ 。

$diff$  函数描述了什么是程序间语法差异性分析，它能够接受两个不同版本的代码，并返回代码间的语法结构差异，也就是该分析所要求的变更集合。

### 4.1.2 分析方法

在本文的兼容性检测方法中，程序间语法差异性分析的主要任务是接受两个不同版本的源代码，并返回代码间的语法结构差异信息。这种结构化的差异信息可以视为补丁的一种，只不过它和常见的采用 Unix diff 工具生成的纯文本补丁文件相比具有更丰富的信息，能够以该语言的合法语法结构的形式对软件变更进行描述。

根据上一节中的相关定义，该分析过程应该具备如下特征：

- 输入为两个不同版本的源代码。即该分析方法应当是针对某种语言的源代码文件进行比对。
- 输出为源代码间的软件变更集合。即该分析方法应当找到这些源代码文件之间发生的变更，并以合适的形式将其输出。
- 每条变更描述对于该语言的合法语法结构的变更。即该分析方法中所找到的变更应当是该源代码实现语言的合法语法结构之间的变化。

除此以外，该分析过程还应当给出更丰富的语法结构信息，以方便后续分析过程的使用，例如：

- 每条变更描述变更前后语法结构的相关信息。例如该语法结构的位置信息，该信息能够指示变更的发生位置等，可用于后续分析过程找到该变更对应的代码位置。
- 每条变更描述了其所属的作用域，即描述了这些语法结构之间的从属关系。该信息描述了变更的作用范围、所属语法结构等，可用于后续分析过程找到某个限定的作用域范围内作用于某种特定语法结构类型的所有变更。

对该分析过程作出这样的要求是为了后续分析过程的方便。由于后续的变更语义影响分析需要我们提供软件变更集合作为输入，而纯文本类型的补丁文件只描述单纯的文本行变更，并不包含语法信息，也就无法从中提取出所需的语法层面的变更信息，因此该类型的程序间文本差异性分析方法是无法满足本文需求的。

在实践过程中，一种比较好的选择是在抽象语法树上进行差异性分析，因为抽象语法树中包括了足够的语法结构信息。

#### 4.1.3 模块设计与实现

本文中的差异性分析模块在实现中采用了 jpf-regression 自带的前置工具 ASTro 来进行程序间语法差异性分析过程。该子模块实现了 *diff* 函数的实际功能，能接受两个不同版本的 Java 源代码文件作为输入，并以 AST 的形式输出变更集合。

ASTro 工具支持对 Java 代码进行比对。它会比对两个源代码文件的抽象语法树，并从中抽取出对应的不同之处，形成语法结构上的变更，并将整个变更前后的 AST 输出，为后续的分析过程提供了所需的变更集合。

该工具将源代码按照抽象语法树的形式进行输出，其最小节点级别为基本块，并提供了丰富的差异性信息，例如两个版本的代码间其对应节点是否发生了变更等，其输出格式为 XML 结构化文档。利用这些输出信息，可以从中提取出需要的程序变更集合，从而进行后续的变更语义影响分析。

在实际使用中，为了满足本文的需要，在将该工具整合进来时进行了一些改进。下面将分别进行阐述。

首先，差异性分析模块对 ASTro 工具的输出结果进行了一定的过滤。受限于 ASTro 工具的具体实现，其输出结果存在着一定的问题，主要包括：

1. 对某些代码文件无法完成差异性分析。这可能是由于有的代码文件过于复杂，超出了其工具的分析能力。
2. 对某些代码文件输出结果不准确，存在误报 (False Positive) 的问题。这可能是受限于工具实现中差异性分析算法的精度，导致将并没有发生变更的语法结构也认为是发生了变更，并进行了分析和输出。

对于第一个问题，由于无法知道该工具的源代码，这里无法解决，但实验结果表明该现象只是极少数。

对于第二个问题，分析其结果可以发现其中存在着误报的情况，即某些代码行并未发生变更，然而工具却报告其发生了诸如移动、先删后增之类的伪变更。同样的，由于无法知道该工具的源代码，这里不能对其差异性分析算法进行修改。因此，在实现过程中差异性分析模块将对改为其输出结果进行专门处理，将这些误报的情况进行过滤，最后保留一个真变更子集合即可。

该过滤算法可以用算法1进行描述。

这里可以归纳证明这种过滤操作的正确性。由于变更对于代码的影响是直接或间接的，对于某次变更语义影响分析的结果  $s$  而言，假设对于其中任意一个受影响的元素  $e_k$ ，其中  $k \in \mathbb{N}$ ，其影响来源可能包括如下几种可能：

1. 其影响仅来源于变更  $c_1$ 。
  - 如果  $c_1$  为真变更，那么删除所有伪变更对于  $e_k$  没有影响。
  - 如果  $c_2$  为伪变更，那么删除所有伪变更会导致  $e_k$  从集合  $s$  中被删除，但此时  $e_k$  本身即为伪影响，集合  $s$  的正确性会得到提高。
2. 其影响来源于多条变更  $c_1, c_2 \dots, c_m$ ，其中  $m \in \mathbb{N}$ 。
  - 假若所有变更均为真变更，那么删除所有伪变更对于  $e_k$  没有影响。
  - 假若来源变更集合中包括某几条伪变更，那么删除所有伪变更之后，仍然存在其他真变更，这些真变更仍然会在变更语义影响分析中导致  $e_k$  被添加到集合  $s$  中，因而也不会使集合  $s$  的正确性下降。
  - 假若所有变更均为伪变更，那么删除所有伪变更会导致  $e_k$  从集合  $s$  中被删除，但此时  $e_k$  本身即为伪影响，集合  $s$  的正确性会得到提高。

可见，本文给出的过滤算法是正确的，它不会导致后续的变更语义影响分析结果  $s$  的正确性降低。

其次，差异性分析模块中完成了分析过程的自动化。

**Algorithm 1** XML 结果过滤算法**Require:**  $c_1 = \text{diff}(v_2, v_1), c_2 = \text{diff}(v_2, v_4)$ **Ensure:** 过滤掉两个变更集合中的相同变更

```
1:  $del_1 \leftarrow \emptyset$ 
2:  $del_2 \leftarrow \emptyset$ 
3: for  $i = 0$  to  $\text{sizeof}(c_1)$  do
4:    $tc_1 \leftarrow c_1[i]$ 
5:   for  $j = 0$  to  $\text{sizeof}(c_2)$  do
6:      $tc_2 \leftarrow c_2[j]$ 
7:     if  $tc_1 == tc_2$  then
8:        $del_1.add(tc_1)$ 
9:        $del_2.add(tc_2)$ 
10:    end if
11:   end for
12: end for
13:  $c_1 \leftarrow c_1.delete(del_1)$ 
14:  $c_2 \leftarrow c_2.delete(del_2)$ 
```

---

在实现差异性分析模块时，该模块采用了 shell 脚本来完成分析过程的自动化，使该模块能够循环地调用 ASTro 进行分析，从而实现对整个软件系统的所有代码文件进行批量化处理。若需要修改该模块的输入信息，只需要修改脚本中对应的输入即可。在这部分工作中，脚本代码主要完成了以下任务：

- 输入数据定位，包括 Java 源代码和编译后的 Class 文件等。该任务为 ASTro 工具提供了输入信息，即不同版本的源代码文件。
- 获取代码文件名，以确定本次分析的对象。该任务选择了某个具体的源代码文件作为输入。
- 实验数据的依赖 JAR 包定位。该任务是为了给出 ASTro 工具需要的额外输入信息，用于源代码编译过程使用。
- 创建输出文件目录。该任务是为了指定 ASTro 工具的输出存放路径。
- 定义 ASTro 的输入参数，包括输入文件位置、输出文件位置、查找路径等。该任务指定了 ASTro 工具运行所需的其他输入信息。
- 调用 ASTro 进行单次分析。该任务即为实际的程序间语法差异性分析过程。

其中 ASTro 工具的使用格式可参考如下，其具体各参数的定义参考表4.1。

---

```

1 ASTDiffer 3/27/2013
2 USAGE: java ASTDiffer –original <file>.java –modified <file>.java
3 –dir <output folder>
4 OPTIONAL: –file <fileName> –ocp <classpath> –mcp <classpath>
5 –oco <outputDir> –mco <outputDir> –cs –xml

```

---

表 4.1 ASTro 参数对照表

| 参数名       | 描述                           | 启用 |
|-----------|------------------------------|----|
| -file     | 分析目标的名字                      | 是  |
| -dir      | 输出路径                         | 是  |
| -ocp      | 旧版本代码的 Classpath             | 是  |
| -mcp      | 旧版本代码的 Classpath             | 是  |
| -original | 旧版本代码的位置                     | 是  |
| -modified | 新版本代码的位置                     | 是  |
| -xml      | 以 XML 格式输出结果                 | 是  |
| -cs       | 以变更脚本 (Change Script) 格式输出结果 | 否  |
| -heu      | 以启发式的方式进行匹配                  | 是  |

该模块中还利用 shell 脚本完成了对后续分析过程的支持，使其能够自动的、批量化的创建影响分析模块所需的配置文件。配置文件为自定义的 JPF 格式，通过类似键值对的方式定义了各项属性的值，其属性描述可以参考表4.2所述。

表 4.2 JPF 属性对照表

| 属性名            | 描述                              |
|----------------|---------------------------------|
| target         | 分析的目标                           |
| sourcepath     | 源代码路径                           |
| rse.ASTResults | ASTro 工具的输出文件位置                 |
| rse.newClass   | 新版本代码的 Class 文件位置               |
| rse.oldClass   | 旧版本代码的 Class 文件位置               |
| rse.dotFile    | jpf-regression 工具的 Dot 格式输出文件位置 |

在使用 shell 脚本调用 ASTro 工具进行分析和输出影响分析模块的配置文件时，由于需要将新版本  $v_2$  作为对比的基准进行分析，因此需要进行相应的配置：

- 令结果  $p_1 = \text{diff}(v_2, v_1)$ ，即将版本  $v_2$  视为“旧版本”，将版本  $v_1$  视为“新版本”。
- 令结果  $p_3 = \text{diff}(v_2, v_4)$ ，即将版本  $v_2$  视为“旧版本”，将在版本  $v_2$  上应用补丁  $p$  后的版本  $v_4$  视为“新版本”。

最后，整个差异性分析模块的工作流程可以参考图4.1。

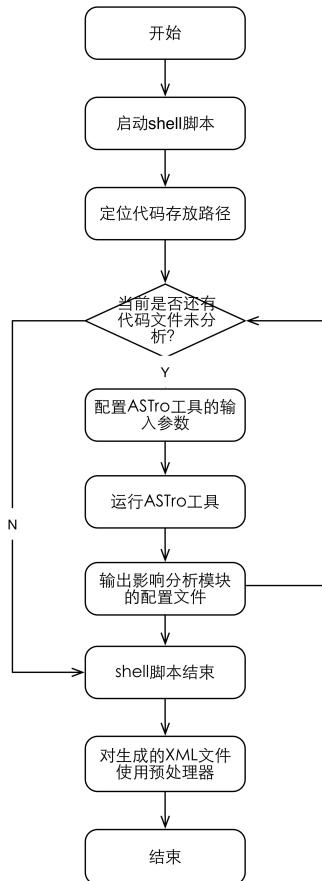


图 4.1 差异性分析模块流程

## 4.2 变更语义影响分析

如前所述，变更语义影响分析的过程主要用于根据上一步中得到的变更集合，找到受到变更集合的语义影响的其他程序语法结构的集合，也就是所谓的变更影响域。该过程主要关注的是语法结构间存在的语义影响。

近年来这方面也有不少比较成熟的工作，因此该模块将直接采用已有的分析工具来完成变更语义影响分析的过程。

### 4.2.1 相关定义

本节主要介绍与变更语义影响分析相关的概念和定义。以下所指的影响都是语义影响。

变更对代码的影响是由于代码之间存在的某种依赖关系而造成的，如果给出不同的依赖关系的定义，那么就会得到不同类型的影响。由此可见，影响关系即

是程序代码间的耦合关系。常见的依赖关系包括控制依赖和数据依赖等。因此，影响关系可以定义如下，其中  $\mathcal{S}$  是某种语言的合法语法结构：

**定义 4.3:**  $im : \mathcal{S} \mapsto \mathcal{S}$ 。该函数描述了程序语法结构之间的影响关系， $\forall s_i, s_j \in \mathcal{S}, i, j \in \mathbb{N}$ ，如果  $im(s_i) = s_j$ ，则说明语法结构  $s_i$  受到  $s_j$  的影响。其中  $im(s_i) = s_j$  当且仅当  $s_i$  依赖于  $s_j$ 。

而变更影响域，也就是变更的语义影响域是指在程序的某个限定的影响范围中，直接或间接受到变更影响的程序语法结构集合。其中影响范围描述了变更的影响传播范围。因此，变更影响域可以较形式化的定义如下，其中  $C$  是某个版本的源代码文件中按照该语言的合法语法结构组织起来的代码， $\mathcal{P}$  是变更集合：

**定义 4.4:**  $impact : \mathcal{P} \times C \mapsto \{\mathcal{S}\}$ 。 $impact(p, v) = \{s_k\}$ ，其中  $im(s_k)$  要么是变更集合  $p$  中所改变的语法结构，要么是  $impact(p, v)$  中的已有元素， $p \subset \mathcal{P}, v \subset C, s_k \subset \mathcal{S}, k \in \mathbb{N}$ 。这里对  $s_k$  选取受限于影响范围和语法结构类型的选择。可见，变更影响域描述了变更集合在代码  $C$  上所影响到的  $\mathcal{S}$  的集合。最后得到的  $\mathcal{S}$  的集合即为直接或间接受到变更影响的语法结构的集合，也就是所谓的变更的语义影响域（即变更影响域）。

在变更影响域的计算过程中，需要考虑到其计算的精度。考虑到上文中的定义，则该过程的计算精度主要受到影响范围和语法结构类型的影响。根据前文中对于  $\mathcal{S}$  的定义，可以将程序中受变更影响的语法结构划分为不同的粒度，从而获得不同程度的影响<sup>[31]</sup>，如类、方法、语句等。而影响范围的粒度则可以划分为：

1. 类间：考虑变更的影响可能传播到其他类（对象）。
2. 方法间：考虑变更的影响可能传播到其他方法内部。
3. 方法内部：考虑变更的影响只在本方法的内部传播。

显然，不同级别的影响范围会对变更语义影响分析的精度产生显著影响。在实践过程中，不同级别的影响范围均可采用不同的语法结构类型，以获得合适精度的变更影响域。

由此可见， $impact$  函数描述了整个变更语义影响分析的过程，该分析过程需要输入代码  $v$  和补丁  $p$ ，其计算结果为在  $v$  中某个范围内受到  $p$  中变更集影响的语法结构集合，即变更影响域。

#### 4.2.2 分析方法

在本文的兼容性检测方法中，该分析过程应当接受两个不同版本代码间的变更集合作为输入，并输出变更集合所对应的语义影响域，也就是所需的变更影响

域。变更语义影响分析的过程可以通过控制流、数据流等信息分析出变更集合中每条变更对其他程序语法结构是否存在影响，并进行闭包计算以找到所有直接或间接接受变更影响的语法结构。

因此，考虑到与前置的程序间语法差异性分析过程和后续的冲突检测过程进行衔接，本文要求该分析过程具备如下特征：

- 接受两个不同软件版本间的变更集合和其中某个版本的代码作为输入。即该分析过程应当分析不同版本间的变更集合对于其中某个版本的影响域。
- 输出该变更集合所对应的影响域。即该分析过程的输出应当是变更集合对于某版本代码的影响域。
- 计算过程中可以指定影响范围和语法结构类型。即该分析过程可以设置分析的精度。
- 具有影响追踪系统，将计算影响域的过程进行记录。即该分析过程需要记录计算过程中涉及到的相关语法结构的影响关系，为后续的冲突检测过程服务，使其能根据影响关系回溯影响的来源。

#### 4.2.3 模块设计与实现

在实现影响分析模块的过程中，本文主要使用了 `jpf-regression` 工具提供的变更语义影响分析算法，并选取了较为简单的分析精度设置，将影响范围限制在方法内部，将受影响的语法结构类型设置为基本块。

该子模块主要实现了 `impact` 函数的实际功能，它接受差异性分析模块中输出的变更集合，计算其变更影响域并将其输出。

`jpf-regression` 是 DiSE 方法在 Java Path Finder 软件框架下的具体实现，提供了限定在方法内的基本块级别的变更语义影响分析算法。在使用过程中，`jpf-regression` 工具接受 Java 格式的源代码和 ASTro 提供的变更集合作为输入，并输出影响域信息至 Dot 文件中。

Dot 是一种采用文本进行描述的图形格式，以该格式输出的实际上是整个源代码的控制流图 CFG(Control Flow Graph)，而变更影响域作为该控制流图承载的额外信息，在图上进行了标注。可见，该模块实际上将影响域信息输出为图形表示，其相关信息主要包括两类：

- 受影响的 CFG 节点。这实际上就是变更影响域中的元素。可见这里受影响语法结构的级别为基本块。
- 节点间的影响关系。这实际上是记录了影响关系的类型，即控制依赖、数据依赖等。

然而 jpf-regression 工具中变更语义影响分析只是其中的一个子模块，主要用于为其后续的 DiSE 分析过程服务。因此影响分析模块将重用 jpf-regression 的代码，并对其进行改造，其主要的变化包括：

1. 修改分析流程。
2. 增加影响追踪系统。
3. 增加错误记录系统。
4. 使其适应大规模批量化分析的需要。
5. 已知 Bug 修复。

下面将分别进行介绍。

影响分析模块中首先对该工具的执行流程进行了修改。jpf-regression 作为 Java Path Finder 框架的一个插件，事实上在使用时需要遵守该框架的约束，有明确的执行流程约定。然而在实践中，该流程约定并不适合于解决本文中的问题，因此本文在实现过程中对该流程进行了一定的修正。

事实上，原执行流程约定，每次分析以源代码文件中的 Main 函数作为入口，探索并分析 Main 函数所调用的其他函数。该流程对于大部分情况而言是具有实际意义的，由于只需考虑 Main 函数及其调用的函数，工具可以减少分析的开销并更快的得出分析结果，而忽略掉其他事实上并未被执行的函数。

然而对于本文所要解决的问题而言，该流程仅适用于部分软件系统，对于某些类型的软件系统而言可能并不适用，例如对 Eclipse JDT Core 项目而言，该项目主要用于为 Eclipse 框架下的其他组件提供服务，其功能将通过库函数的形式来提供。对于这类以库函数形式对外提供服务的源代码而言，它们既并不存在入口函数，也无法预知到底会有哪些函数会被外界所调用。由此可见，该模块应当在分析过程中覆盖其所有函数，以保证结果的完整性和正确性。

本文对于流程的修改如下所述。首先在 jpf-regression 工具的启动流程中去掉了图4.2中所示的使用 JPF 框架进行启动的方式，改为直接循环调用 jpf-regression 的核心功能，该新流程中采用 RunAll 类和 Runjpf 类来实现大规模分析的功能，其具体描述参见后文说明。其次，如图4.3所示，该模块中将待计算的方法集合由 Main 函数的调用函数集合更改为源代码中所有函数的集合，并在相应的影响集合计算过程中增加了影响追踪系统。

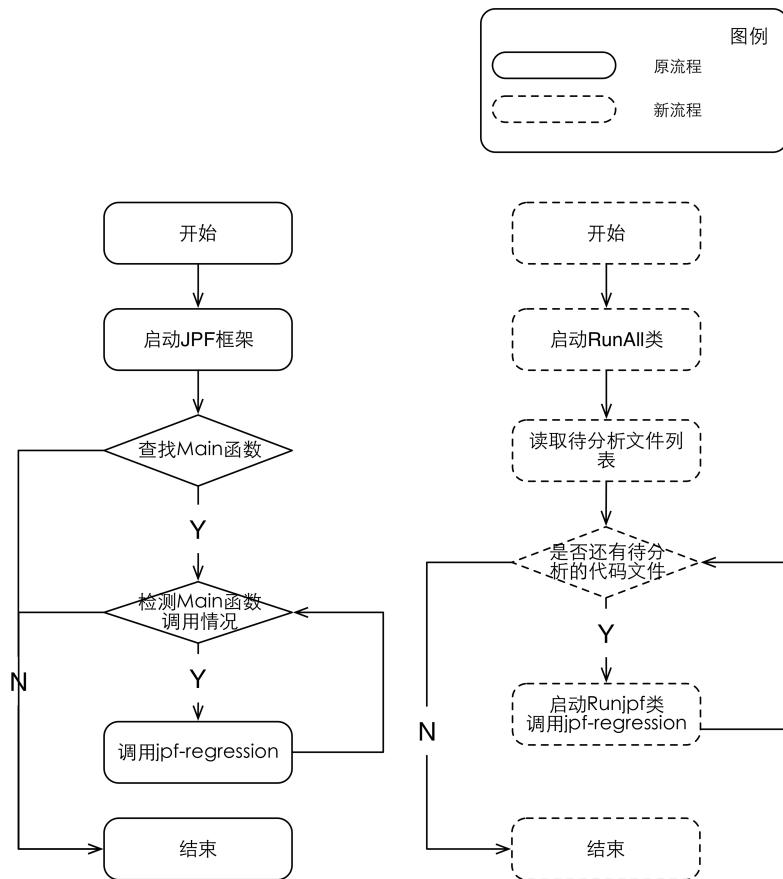


图 4.2 jpf-regression 启动流程及变化

影响分析模块中对影响追踪系统进行了实现。由于在后续的冲突判定模块中，对于找到的可能发生冲突的位置，需要输出其影响的来源以进行人工分析和判定，因此该模块将在变更语义影响分析的阶段加入影响追踪系统，以便记录下变更影响的轨迹，根据这些信息为后续的分析过程提供便利。

影响追踪系统需要存储变更影响域计算过程中找到的程序语法结构间的影响关系。由于影响的来源主要有两类，即：

- 控制依赖。即某个语法结构的执行依赖于某条件语句的选择。
- 数据依赖。即某个语法结构中使用到了其他语法结构中定义或赋值的变量。

因此，本文实现了 Dependency 类族来描述该影响关系，参见图4.4。该类中使用了二元组 depend 来存储影响关系，并使用了多态机制来区分影响关系的类型。并且 Dependency 类族均重写了 hashCode() 方法和 equals() 方法，以便能够使用集合操作。影响追踪系统中具体使用到的数据结构可以参考表4.3。在实现过程中，将影响关系作为控制流中承载的额外信息输出即可。

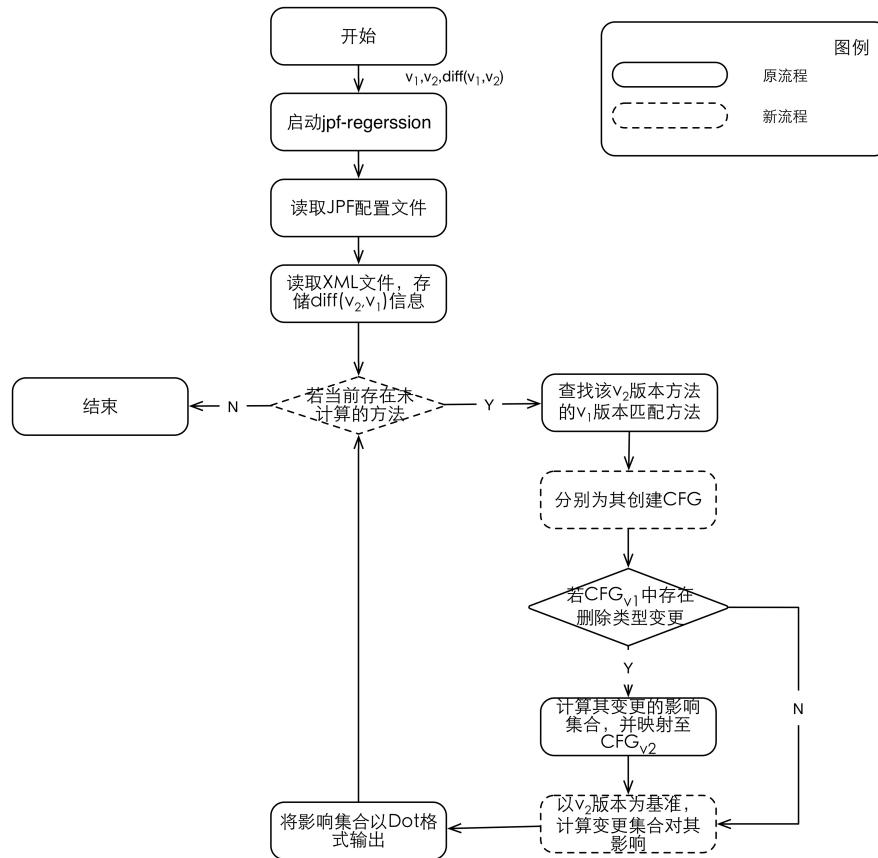


图 4.3 jpf-regression 原流程及变化

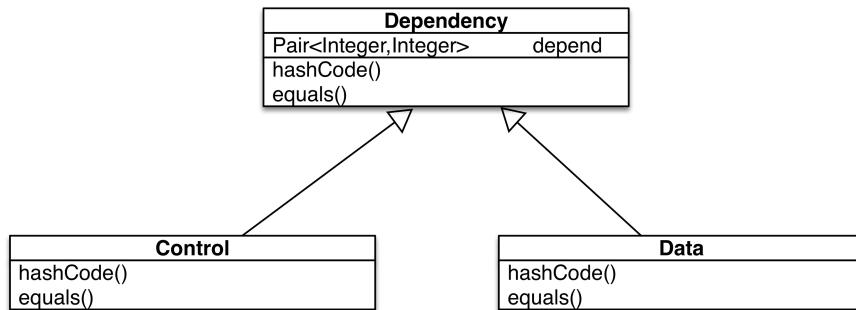


图 4.4 Dependency 类族

表 4.3 影响关系数据结构

| 数据类型                          | 数据结构       | 用途             |
|-------------------------------|------------|----------------|
| Dependency                    | dependency | 影响关系           |
| Control                       | dependency | 控制依赖影响关系       |
| Data                          | dependency | 数据依赖影响关系       |
| Map<Integer, Set<Dependency>> | depend     | 存储计算过程中的全部影响关系 |

影响分析模块还令分析工具适应了大规模分析过程的需要。jpf-regression 工具自身只支持每次对单对源代码文件进行分析，然而考虑到软件系统在版本更新时可能有大量的源代码文件发生了变更，该模块需要具备大规模、批量化分析的能力以应对这种情况。为此，在实现过程中可以循环调用 jpf-regression 工具中的单次分析过程，并在上层进行封装，使大规模分析过程对于用户而言是透明的。

为了实现大规模分析过程，还需要修改工具中的某些实现细节，包括输出文件命名格式、异常处理方式等，并增加一定的数据统计能力。

在进行大规模分析的时候，工具本身自定义的输出文件命名格式需要进行修改。在原单次分析过程中，输出文件被直接命名为被分析的方法名。对于单次分析过程而言，这种设计是合适的，然而在大规模分析的情况下，由于可能存在的某些现象，例如函数重载和不同版本间的代码其方法可能无法一一匹配等，该命名方式就需要进行一定的优化。在这种情况下，可以采用一种新的命名格式：`MethodName+HashCode(MethodName)+ExtensionName`。

其中，`MethodName` 为方法名，`HashCode(MethodName)` 是利用 Java 语言中的 `hashCode` 方法对 `MethodName` 字符串进行计算所得到的后缀名，以保证整个文件名的唯一性。最后的 `ExtentionName` 即为文件扩展名。

大规模分析过程中还需要修改原有的异常处理模式。由于原有的 jpf-regression 工具只进行单次分析过程，因此一旦在运行过程中遇到问题，就会抛出异常并终止分析过程的运行。然而在实际情况中由于需要进行大规模的分析过程，如果仅仅由于单个文件的分析出错就终止整个分析过程，就会造成极大的时间和计算资源的浪费。

因此影响分析模块中修改了该工具的异常处理方式，使其在单次分析过程中如果遇到问题，只抛出异常而并不终止整个程序的运行，即采取了继续往下执行以分析其他文件的策略。然而分析中出现的异常是确实存在的，为了不丢失这类错误信息，本文为工具添加了错误记录系统，以记录每个单次分析过程中遇到的问题。

本文对程序运行过程中可能出现的报错情况进行了分类，并设计了专门的错误统计类以专门别类的对错误情况进行记录和统计。该类的实现可以参考图4.5。其中使用到的数据结构可以参考表4.4的说明。

表 4.4 错误记录数据结构

| 数据类型                | 数据结构             | 用途                |
|---------------------|------------------|-------------------|
| String              | xml_not_found    | 错误: XML 文件未找到     |
| String              | class_not_found  | 错误: Class 文件未找到   |
| String              | interface_class  | 错误: 接口类           |
| String              | null_class       | 错误: 类中无具体实现(如抽象类) |
| Set<String>         | precise_analysis | 记录有多少方法在影响计算过程中出错 |
| Map<String, String> | class_error      | 记录有哪些类出现了哪些错误     |
| void                | print()          | 输出错误记录            |

| ErrorCount          |                  |
|---------------------|------------------|
| String              | xml_not_found    |
| String              | class_not_found  |
| String              | interface_class  |
| String              | null_class       |
| String              | className        |
| String              | methodName       |
| Set<String>         | precise_analysis |
| Map<String, String> | class_error      |
| print()             |                  |

图 4.5 ErrorCount 类

大规模分析过程中，由于实验数据量非常庞大，无法像单次分析过程中那样进行人工的检查和实验结果分析。因此，影响分析模块中还增加了数据统计模块，使得程序具备一定的自动化分析实验结果的能力。

大规模分析的相关类实现可以参考图4.6和图4.7。其中涉及到的数据结构及其说明参考表4.5和表4.6。在实现中，RunAll 会读取所有待分析的文件名并找到其配置文件路径，然后循环调用 Runjpf 对象进行单次分析过程。

| RunAll                          |           |
|---------------------------------|-----------|
| Set<String>                     | filenames |
| Set<ErrorCount>                 | errors    |
| void readFileName(String path)  |           |
| static void main(String[] args) |           |
| void bakDot(String p)           |           |
| void deleteDot(String p)        |           |

图 4.6 RunAll 类

表 4.5 RunAll 数据结构

| 数据类型            | 数据结构                      | 用途            |
|-----------------|---------------------------|---------------|
| Set<String>     | filenames                 | 存储待分析的文件名     |
| Set<ErrorCount> | errors                    | 存储每次分析中出现的错误  |
| void            | readFileName(String path) | 从文件中读取待分析的文件名 |
| static void     | main(String[] args)       | 实现大规模分析过程     |
| void            | bakDot(String p)          | 备份分析结果        |
| void            | deleteDot(String p)       | 删除分析结果        |

表 4.6 Runjpf 数据结构

| 数据类型        | 数据结构                     | 用途                                      |
|-------------|--------------------------|---|
| String      | file                     | 待分析文件名                                  |
| String      | old_to_new               | $impact(diff(v_2, v_1), v_2)$ 过程的配置文件位置 |
| String      | new_to_patch             | $impact(diff(v_2, v_4), v_2)$ 过程的配置文件位置 |
| void        | change(String file)      | 改变当前需要读取的配置文件                           |
| ErrorCount  | run_otn(String filename) | 运行 $impact(diff(v_2, v_1), v_2)$ 过程     |
| ErrorCount  | run_ntp(String filename) | 运行 $impact(diff(v_2, v_4), v_2)$ 过程     |
| static void | main(String[] args)      | 实现单次分析过程                                |

| Runjpf      |                          |
|-------------|--------------------------|
| String      | file                     |
| String      | old_to_new               |
| String      | new_to_patch             |
| void        | change(String file)      |
| ErrorCount  | run_otn(String filename) |
| ErrorCount  | run_ntp(String filename) |
| static void | main(String[] args)      |

图 4.7 Runjpf 类

最后，影响分析模块在使用 jpf-regression 工具的过程中还修复了某些 Bug。由于在运行过程中该工具暴露出了某些 Bug，且这些 Bug 或多或少的导致了分析结果的正确性和精度降低，因此，在影响分析模块在实现过程中对其中某些可修复的 Bug 进行了修复。目前已知的 Bug 及其修复情况可以参见表4.7。

表 4.7 Bug 报告

| Bug                                      | 频率及危害 | 修复 |
|--|-------|----|
| 内部类无法进行方法匹配                              | 低, 小  | 否  |
| 只有单个版本代码中存在某个方法时无法进行方法匹配                 | 低, 小  | 是  |
| 将 $CFG_{v1}$ 的影响集合映射到 $CFG_{v2}$ 时判断条件出错 | 高, 大  | 是  |
| 调用依赖 JAR 包 jpf_guided_test 出错            | 低, 小  | 否  |
| 调用依赖 JAR 包 jpf_symbc 出错                  | 低, 小  | 否  |

### 4.3 本章小结

本章中主要介绍了软件变更影响域分析方法和其对应的模块设计与实现过程。章节4.1中介绍了程序间差异性分析方法和其对应的模块设计与实现。章节4.2中介绍了变更语义影响分析方法和其对应的模块设计与实现。

## 第 5 章 软件变更冲突检测方法

本章中主要介绍软件变更冲突检测方法，包括具体的检测算法、相应的模块设计与实现等。

软件变更冲突检测主要用于对变更影响域之间是否发生冲突进行判定。本文中实现了简单的自动分析算法，通过判断变更影响域之间是否存在重叠来找到可能存在冲突的代码位置，并结合人工分析来完成最后的判定。

下面对该冲突检测方法和其对应模块的设计与实现进行具体的介绍。

### 5.1 相关定义

为了进行变更的冲突检测，需要知道什么是冲突。下文中的冲突都指语义冲突。

根据前文中的讨论，冲突实际上是指两个补丁所对应的不同变更集合之间存在某两条互斥的变更，即这两条变更的影响域发生了重叠。因此，冲突可以较形式化的定义如下，其中  $C$  是某个版本的源代码文件中按照该语言的合法语法结构组织起来的代码， $\mathcal{P}$  是变更集合， $impact$  是求解影响域的函数：

**定义 5.1：**  $conflict : \mathcal{P} \times \mathcal{P} \mapsto T, F$ 。 $\forall p_i, p_j \subset \mathcal{P}$ ，如果  $impact(p_i, v) \cap impact(p_j, v) \neq \emptyset$ ，那么就有  $conflict(p_i, p_j) = T$ ，其中  $v \subset C, i, j \in \mathbb{N}$ 。即如果两个补丁分别对应的变更集合在某相同版本代码上的变更影响域之间产生了交集，那么就认为补丁间发生了冲突。

综上所述，冲突是由于来自两个变更集合的某两条不同变更分别直接或间接的影响到了某个相同的语法结构而造成的，可见  $conflict$  函数描述了变更冲突检测的过程，它接受两个变更集合作为输入，并根据变更影响域计算其是否发生了冲突。

### 5.2 分析方法

如前所述，冲突检测的过程即为  $conflict$  函数的具体实现。该分析过程将对比两个补丁间的变更影响域，通过判断变更影响域间是否重叠来检测其兼容性。

理论上来讲，由于重叠部分的代码显然是可能会发生冲突的，这种简单比对即可发现两个补丁间的兼容性问题。然而在具体实现中，受限于工具的精度，冲突检测方法往往不能达到理论上的准确度，而可能产生误报等情况。

如果重叠不存在，则补丁间的语义没有发生相互覆盖，因而不存在冲突。如果存在重叠，则对于重叠部分的代码而言，判断该位置是否确实发生了冲突需要人工分析的辅助。因为该代码位置的冲突判定与补丁的变更目的密切相关，而代码中无法直接获取到这种信息，因此只有依靠外界的干预来判定这部分代码是否确实存在冲突。

人工分析过程不仅需要知道哪部分代码出现了重叠，还需要知道是哪些变更影响到了这部分代码。因此冲突检测过程需要具有影响回溯的能力。由于变更影响域分析中的影响追踪系统会记录其分析过程的轨迹并存储程序语法结构间的影响关系，因此只需在冲突检测时对重叠部分的代码进行回溯即可追踪到具体的软件变更。

该冲突检测过程可以参考算法2。

---

**Algorithm 2** 冲突检测

---

**Require:**  $s_1 = impact(diff(v_2, v_1), v_2)$ ,  $s_2 = impact(diff(v_2, v_4), v_2)$

**Ensure:**  $conflict(diff(v_2, v_1), diff(v_2, v_4))$

```
1: if  $s_1 = \emptyset \vee s_2 = \emptyset$  then
2:     result  $\leftarrow False$ 
3: else
4:      $s \leftarrow s_1 \cap s_2$ 
5:     if  $s = \emptyset$  then
6:         result  $\leftarrow False$ 
7:     else
8:         result  $\leftarrow Manual\_analysis(s_1, s_2)$ 
9:     end if
10: end if return result
```

---

### 5.3 模块设计与实现

冲突判定模块主要需要实现  $conflict$  函数的实际功能。目前根据上文所述的冲突检测算法实现了较为简单的自动分析过程，更精确的分析结果需要人工分析过程的辅助。

考虑到该模块需要实现接受影响域分析模块的输入，并完成冲突检测的功能，该模块的核心任务应当包括：

- **输入：**读取软件变更影响域分析过程的输出，即该模块需要接受两个变更影响域作为其输入。

- 输出：找到可能发生冲突的代码位置，并输出其影响来源。即该模块需要输出变更影响域重叠处的代码位置（即可能发生冲突的位置）及其相关的影响关系。
- 影响域计算：存储读取的影响域信息，并计算其是否发生重叠。即该模块需要实现上文提出的冲突检测算法。

该模块的实现过程主要参考了算法2的描述。由于其输入为影响域模块的输出，因此它在输出时可以参照其输入给出类似形式的结果，即将重叠位置和其相关的影响关系作为额外的信息输出到控制流图中，以供人工分析使用。人工分析的过程主要是参考输出的部分控制流中，被标注为可能发生冲突的节点是否确实发生了冲突。

冲突判定模块中的类实现可以参考图5.1。其中 impactSet 类用于存储影响域，DotNode 和 DotEdge 用于存储从 Dot 文件中读取到的节点和边的信息，这两个类采用了多态机制来存储节点和边的类型信息。Diff 类是从 Dot 文件中读取影响域并计算其重叠与否的类。相关的数据结构说明参考表5.1。

表 5.1 Diff 数据结构

| 数据类型        | 数据结构  | 用途                    |
|-------------|---|-----------------------|
| static void | main(String[] args)                               | 实现冲突分析过程              |
| void        | analyzeDot(String path, impactSet now)            | 读取 Dot 文件，将影响域存储于 now |
| void        | readFileName(String path)                         | 读取待分析的文件名             |
| void        | deleteDot(String p)                               | 删除输出                  |
| void        | computeImpact(impactSet s, impactSet s1)          | 计算重叠                  |
| void        | output(Set<Integer> intersection, impact now)     | 输出冲突和相关控制流            |
| void        | printNode(BufferedWriter writer, DotNode node...) | 写控制流节点                |
| void        | WriteEdge(Set<Integer> impacted...)               | 写控制流边                 |

表 5.2 impactSet 数据结构

| 数据类型                       | 数据结构         | 用途               |
|----------------------------|--------------|------------------|
| String                     | path         | 该影响域对应 Dot 文件的位置 |
| Set<Integer>               | allLoc       | 存储影响域中的元素        |
| Set<DotNode>               | nodes        | Dot 文件中的控制流节点    |
| Set<DotEdge>               | edges        | Dot 文件中的控制流边     |
| Map<Integer, Set<Integer>> | lineToNodeId | 从控制流边到节点 ID 的映射  |
| Map<Integer, DotNode>      | idToNode     | 从节点 ID 到节点的映射    |
| Map<Integer, Set<DotEdge>> | relation     | 从节点 ID 到边的映射     |

表 5.3 DotNode 数据结构

| 数据类型    | 数据结构    | 用途          |
|---------|---------|-------------|
| Integer | begin   | 该基本块的起始行号   |
| Integer | end     | 该基本块的结束行号   |
| Integer | id      | 该节点的 ID     |
| boolean | changed | 该节点是否属于变更集合 |

表 5.4 DotEdge 数据结构

| 数据类型    | 数据结构  | 用途      |
|---------|-------|---------|
| Integer | begin | 该边的起始节点 |
| Integer | end   | 该边的结束节点 |

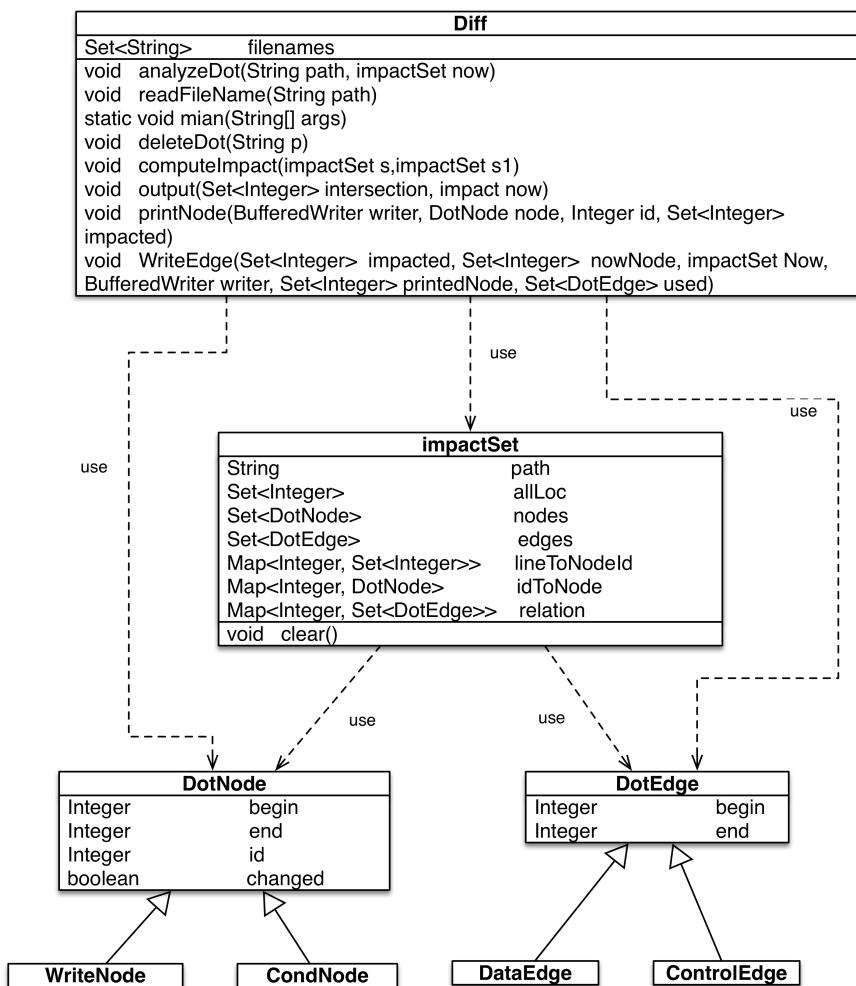


图 5.1 Diff 类族

综上所述，该模块的实际工作流程可以参考图5.2。

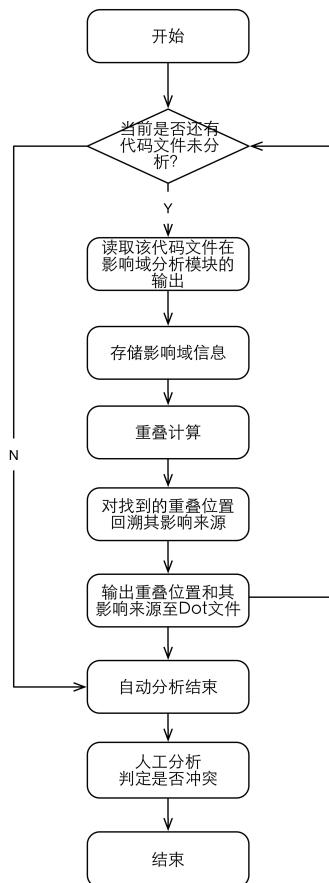


图 5.2 冲突检测模块流程

## 5.4 本章小结

本章中主要介绍了软件变更冲突检测方法和其对应的模块设计与实现。章节5.1中介绍了冲突检测方法的相关定义。章节5.2中介绍了冲突检测方法和其算法描述。章节5.3中介绍了冲突检测方法对应的工具模块其设计与实现。

## 第6章 实验结果与分析

本章主要介绍如何对补丁兼容性检测工具进行实验，并给出了实验结果和分析。

### 6.1 实验设计

考虑到本文讨论的软件补丁兼容性检测问题是一个工业界中常见的问题，广泛存在于各类项目的开发、维护过程中，因此在对该检测工具进行实验时，应当选择工业界中常见、常用的中大型项目，使得实验结果具有较强的说服力，能够说明本文所提出的检测方法是否能切实解决工业界中面对的实际问题。

可见，实验应当具备如下目的：

- 测试检测工具是否能够对中大型项目成功进行分析。该项可以说明本文方法的可用性，即是否能够对工业界的实际项目进行分析。
- 测试检测工具是否能够找到补丁的兼容性问题，其检测结果是否正确。该项可以说明本文方法的正确性，即是否能够正确地检测到的补丁间的语义冲突。
- 测试检测工具能找到多少兼容性问题。该项可以说明本文方法的实用性，即是否能够完全覆盖所有的补丁间冲突。

因此，本章中应当进行满足上述要求的实验，并对其结果给出相应的量化表述。

根据上面的需求，本章中的实验过程可以设计如图6.1所示。



图 6.1 实验设计

下面就本文中所采用的实验平台，对其配置说明如下：

- 操作系统： Mac OS X 10.9
- CPU： 2.4 GHz Intel Core i5
- 内存： 8 GB 1600 MHz DDR3
- 硬盘： 251 GB APPLE SSD SM0256F Media

## 6.2 实验案例

本文将采用 Eclipse JDT Core 项目和其相关的补丁作为实验案例，以测试检测工具的可靠性和可用性等。JDT Core 是 Eclipse 工具的基础组件，用于支持 Java 语言的编译等功能。该项目的相关信息可以参见表6.1。

表 6.1 Eclipse JDT Core

| 信息  | 描述       |
|-----|----------|
| 语言  | Java     |
| 文件数 | 约 1200 个 |
| 代码量 | 约 45W 行  |

Groovy-Eclipse 是一个 Eclipse 插件集合，用于为 Groovy 语言提供 Eclipse 的工具支持。通过 Groovy-Eclipse 提供的 Eclipse JDT Core 功能增强补丁，Groovy 语言的代码可以被 Eclipse JDT Core 编译为 Java 字节码，从而在 JVM 上运行。这一特性使得 Groovy 可以调用其他 Java 语言编写的库，从而大大提升了其可用性。本文选择应用该功能增强补丁后的 Eclipse JDT Core 作为实验中版本  $v_3$  的来源。

Eclipse JDT Core 项目截止目前已推出至发行版 4.4.2，本文从中选择了若干个发行版作为实验中版本  $v_1$  和  $v_2$  的来源。实验数据的具体选择可以参考表6.2，对各版本的具体说明可以参考表6.3。由于版本  $v_4$  是由版本合并而来，因此在表6.2中不予以列出。

根据表6.2，本文共选择了 7 个发行版本作为版本  $v_2$  的来源。也就是说，实验将以固定的版本  $v_1$  和  $v_3$  为基准，并选择不同的实验数据作为版本  $v_2$  来进行实验。

表 6.2 实验数据

| 代码                      | 发行版本  | 版本对照  |
|-------------------------|-------|-------|
| Eclipse JDT Core        | 4.3.2 | $v_1$ |
| Groovy-Eclipse JDT Core | 4.3.2 | $v_3$ |
| Eclipse JDT Core        | 4.4   | $v_2$ |
| Eclipse JDT Core        | 4.4.2 | $v_2$ |
| Eclipse JDT Core        | 4.3   | $v_2$ |
| Eclipse JDT Core        | 4.3.1 | $v_2$ |
| Eclipse JDT Core        | 4.2   | $v_2$ |
| Eclipse JDT Core        | 4.2.1 | $v_2$ |
| Eclipse JDT Core        | 4.2.2 | $v_2$ |

表 6.3 版本对照表

| 版本 描述 |  |
|-------|--|
| $v_1$ | 旧版本                                      |
| $v_2$ | 新版本                                      |
| $v_3$ | 应用补丁 p 后的旧版本                             |
| $v_4$ | 版本 $v_2$ 和版本 $v_3$ 合并后版本，相当于应用补丁 p 后的新版本 |

## 6.3 实验结果与分析

由于检测方法由多个步骤组成，因此实验是分步进行的，本节将给出每个步骤的输入输出数据，并对其结果加以分析。

### 6.3.1 版本迁移

如章节3.2所述，实验首先需要完成将补丁 p 应用于版本  $v_2$  的过程。该过程采用 git 工具进行版本合并，并使用 Beyond Compare 工具解决合并时可能出现的语法冲突。

根据上节中所选择的实验案例，补丁的版本迁移过程可以参考图6.2。该过程以 Eclipse JDT Core 发行版 4.3.2 作为旧版本  $v_1$ ，以 Groovy-Eclipse JDT Core 发行版 4.3.2 作为版本  $v_3$ ，以其他 Eclipse JDT Core 发行版作为版本  $v_2$ ，并为不同的版本  $v_2$  创建了独立分支，用于实现各自的版本合并过程。

版本合并过程中检测到的待解决冲突数量可以参见表6.4。通过 Beyond Compare 工具，这些冲突都能够得到解决。

根据图6.3可以发现，冲突最少的是版本 4.3.x，几乎为 0%，这可能是由于版本 4.3 到版本 4.3.2 的升级过程改动较少而造成的。而对于版本 4.4.x 来说，冲突百分比甚至高达 20% 至 30%，这可能是由于版本升级较大的缘故。

表 6.4 版本合并结果

| 代码               | 发行版本  | 冲突文件数量 | 所有文件 |
|------------------|-------|--------|------|
| Eclipse JDT Core | 4.4   | 313    | 1285 |
| Eclipse JDT Core | 4.4.2 | 596    | 1281 |
| Eclipse JDT Core | 4.3   | 10     | 1209 |
| Eclipse JDT Core | 4.3.1 | 10     | 1209 |
| Eclipse JDT Core | 4.2   | 50     | 1205 |
| Eclipse JDT Core | 4.2.1 | 49     | 1205 |
| Eclipse JDT Core | 4.2.2 | 49     | 1205 |

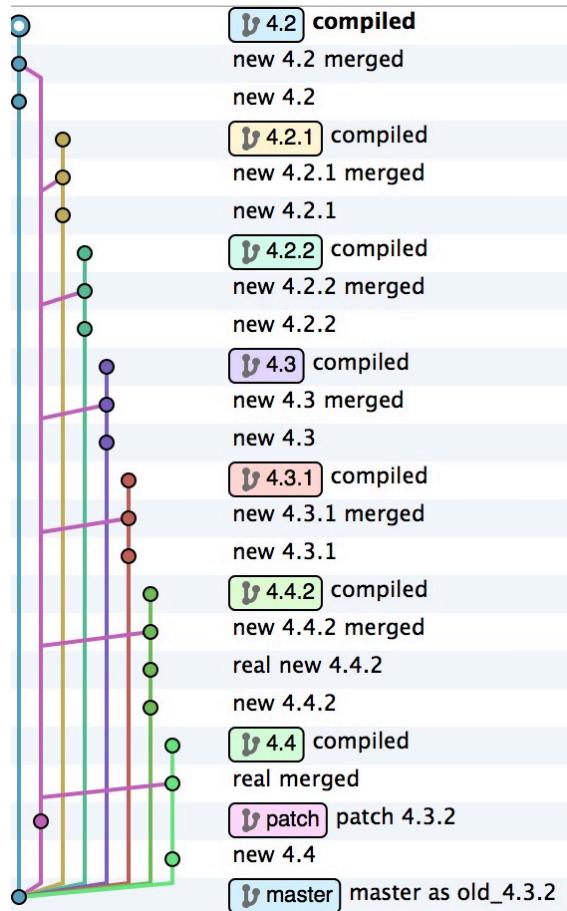


图 6.2 git 版本合并

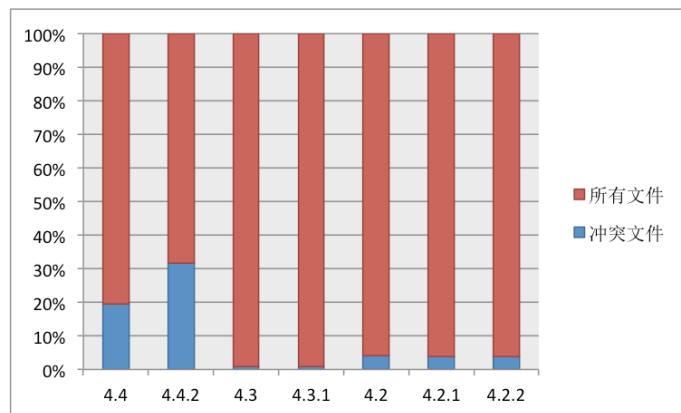


图 6.3 版本合并结果

在实验过程中，由于后续的影响域分析模块需要提供 Java 代码编译后产生的 Class 文件，合并后的版本  $v_4$  还需要编译。实验结果表明，绝大多数的文件都能够正常编译通过，只有极少部分文件由于合并出错等原因而无法编译通过。该部分数据可以参考表6.5。同样，从图6.4可以看出，编译失败的文件数量所占的百分比很低。

参考图6.5，对比编译过程中的失败数据与版本合并中的冲突数据，可见二者的变化过程是较为吻合的。

表 6.5 编译结果

| 代码               | 发行版本  | 编译失败文件 | 所有文件 |
|------------------|-------|--------|------|
| Eclipse JDT Core | 4.4   | 23     | 1285 |
| Eclipse JDT Core | 4.4.2 | 18     | 1281 |
| Eclipse JDT Core | 4.3   | 0      | 1209 |
| Eclipse JDT Core | 4.3.1 | 1      | 1209 |
| Eclipse JDT Core | 4.2   | 4      | 1205 |
| Eclipse JDT Core | 4.2.1 | 4      | 1205 |
| Eclipse JDT Core | 4.2.2 | 1      | 1205 |

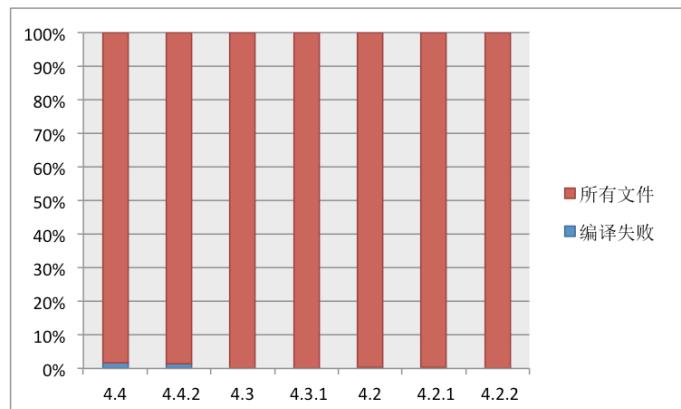


图 6.4 编译结果

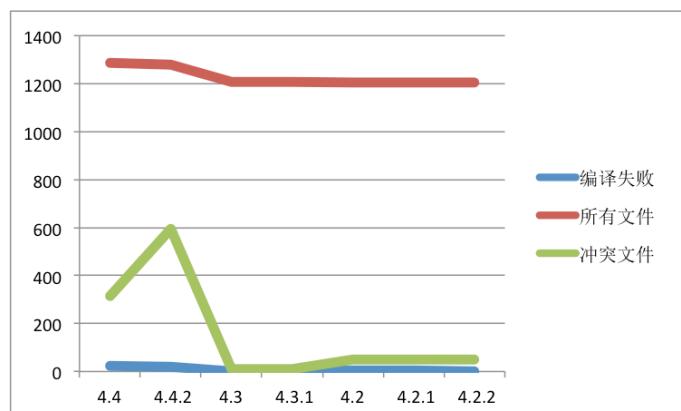


图 6.5 版本合并总结果

综上所述，本文的版本迁移过程是简单且有效的。

### 6.3.2 影响域分析模块

由于整个影响域分析模块可以分为差异性分析模块和影响分析模块两个子模块，下面将分别阐述这两个子模块的实验结果并进行分析。

#### 6.3.2.1 差异性分析模块

在差异性分析模块中，本文主要关注过滤算法的结果以及能够成功完成程序间语法差异性分析的文件数量。

过滤算法的实验结果如表6.6和图6.6所示。从结果来看，本文的过滤算法对原有的 ASTro 工具的直接输出结果进行了有效的过滤，有将近 10% 的文件出现了伪变更，并被成功过滤。

表 6.6 差异性分析模块结果

| 代码               | 发行版本  | 过滤数 | $diff(v_2, v_1)$ 文件数 | $diff(v_2, v_4)$ 文件数 |
|------------------|-------|-----|----------------------|----------------------|
| Eclipse JDT Core | 4.4   | 91  | 1088                 | 1172                 |
| Eclipse JDT Core | 4.4.2 | 94  | 1097                 | 1178                 |
| Eclipse JDT Core | 4.3   | 110 | 1126                 | 1124                 |
| Eclipse JDT Core | 4.3.1 | 110 | 1126                 | 1123                 |
| Eclipse JDT Core | 4.2   | 99  | 1115                 | 1117                 |
| Eclipse JDT Core | 4.2.1 | 99  | 1115                 | 1116                 |
| Eclipse JDT Core | 4.2.2 | 99  | 1115                 | 1116                 |

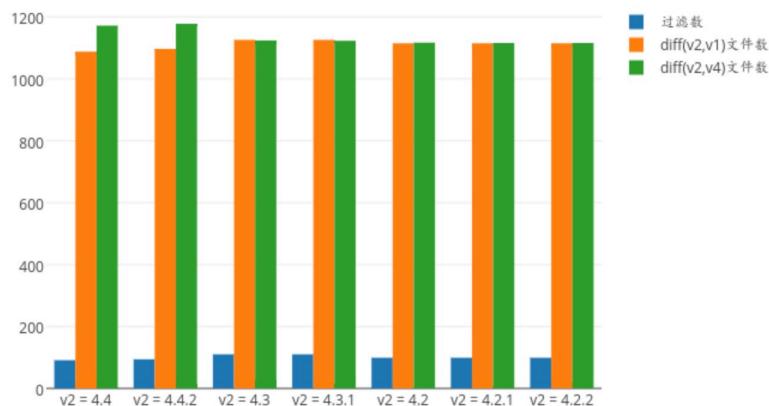


图 6.6 差异性分析模块结果

整个差异性分析模块的输出结果如表6.7和表6.8所述。更直观的表示可以参考图6.7和6.8。结果表明，绝大多数的文件都能够成功的进行程序间语法差异性分析，该差异性分析模块是可用的，其结果的正确性被过滤算法提高了。

表 6.7 差异性分析模块结果

| $v_1$ | $v_2$ | $diff(v_2, v_1)$ 文件数 | $v_2$ 文件 | $v_1$ 文件 |
|-------|-------|----------------------|----------|----------|
| 4.3.2 | 4.4   | 1088                 | 1272     | 1200     |
| 4.3.2 | 4.4.2 | 1097                 | 1272     | 1200     |
| 4.3.2 | 4.3   | 1126                 | 1200     | 1200     |
| 4.3.2 | 4.3.1 | 1126                 | 1200     | 1200     |
| 4.3.2 | 4.2   | 1115                 | 1196     | 1200     |
| 4.3.2 | 4.2.1 | 1115                 | 1196     | 1200     |
| 4.3.2 | 4.2.2 | 1115                 | 1196     | 1200     |

表 6.8 差异性分析模块结果

| $v_4$    | $v_2$ | $diff(v_2, v_4)$ 文件数 | $v_2$ 文件 | $v_4$ 文件 |
|----------|-------|----------------------|----------|----------|
| 基于 4.4   | 4.4   | 1172                 | 1272     | 1278     |
| 基于 4.4.2 | 4.4.2 | 1178                 | 1272     | 1274     |
| 基于 4.3   | 4.3   | 1124                 | 1200     | 1202     |
| 基于 4.3.1 | 4.3.1 | 1123                 | 1200     | 1202     |
| 基于 4.2   | 4.2   | 1117                 | 1196     | 1198     |
| 基于 4.2.1 | 4.2.1 | 1116                 | 1196     | 1198     |
| 基于 4.2.2 | 4.2.2 | 1116                 | 1196     | 1198     |

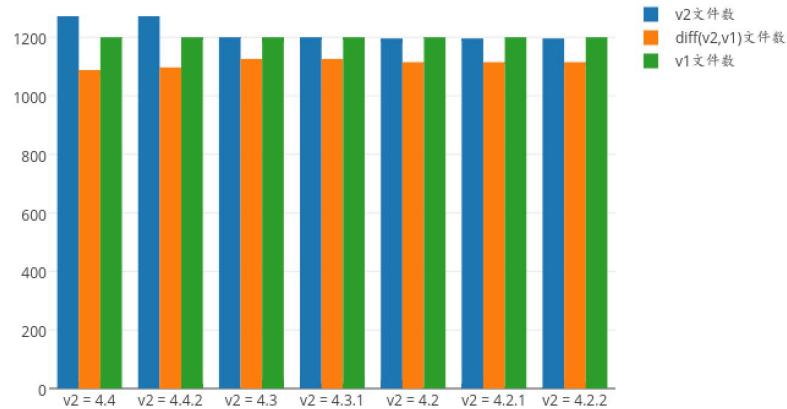


图 6.7 差异性分析模块结果

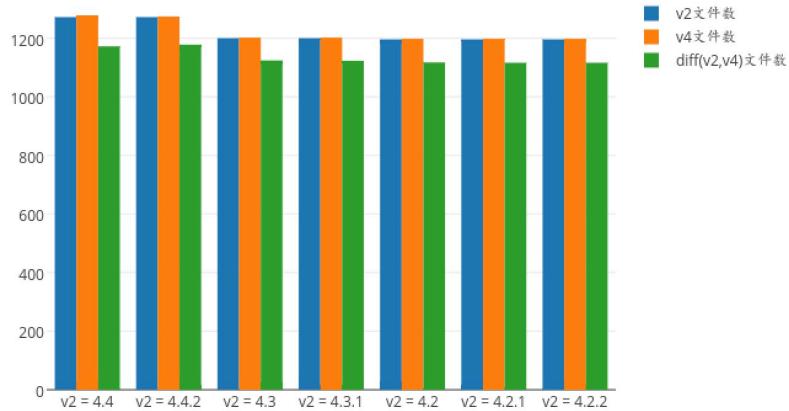


图 6.8 差异性分析模块

### 6.3.2.2 影响分析模块

在影响分析模块中，本文主要关注能够成功进行分析的文件数量。

对  $impact(diff(v_2, v_1), v_2)$  过程而言，应用影响分析模块后，分析结果如表6.9所述。从结果中可以看出，4.3.x 和 4.2.x 系列版本的成功分析数量略多于 4.4.x 系列的版本。

表 6.9 影响分析模块结果

| 代码               | $v_2$ | 成功分析数 |
|------------------|-------|-------|
| Eclipse JDT Core | 4.4   | 881   |
| Eclipse JDT Core | 4.4.2 | 892   |
| Eclipse JDT Core | 4.3   | 930   |
| Eclipse JDT Core | 4.3.1 | 930   |
| Eclipse JDT Core | 4.2   | 918   |
| Eclipse JDT Core | 4.2.1 | 923   |
| Eclipse JDT Core | 4.2.2 | 924   |

综合考虑其输入数据的数量进行对比，其结果如图6.9所示。可以发现虽然 4.4.x 系列版本中代码文件的数量增加了，但其能成功分析的文件数量反而减少了。

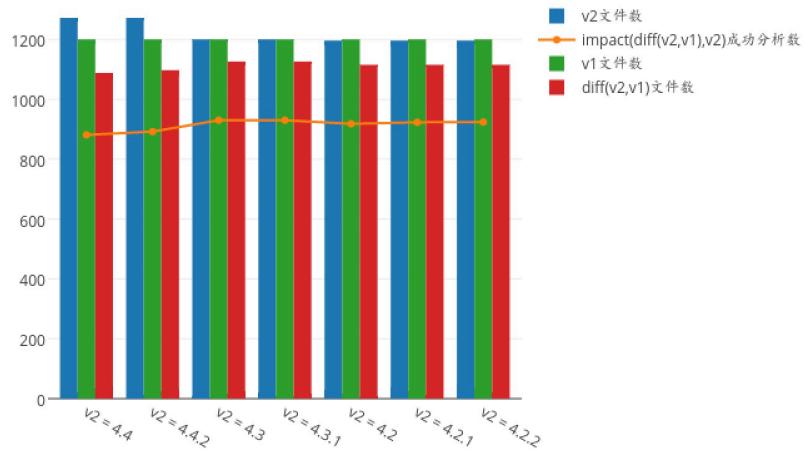


图 6.9 影响分析模块

对  $impact(diff(v_2, v_4), v_2)$  过程而言, 应用影响分析模块后, 分析结果如表6.10所述。该过程中能够成功分析的文件数量与  $impact(diff(v_2, v_1), v_2)$  相类似。

表 6.10 影响分析模块结果

| 代码               | $v_2$ | 成功分析数 |
|------------------|-------|-------|
| Eclipse JDT Core | 4.4   | 881   |
| Eclipse JDT Core | 4.4.2 | 892   |
| Eclipse JDT Core | 4.3   | 930   |
| Eclipse JDT Core | 4.3.1 | 925   |
| Eclipse JDT Core | 4.2   | 916   |
| Eclipse JDT Core | 4.2.1 | 924   |
| Eclipse JDT Core | 4.2.2 | 928   |

综合考虑其输入数据的数量进行对比, 其结果如图6.10所示。该图中显示的成功分析的文件数量其走势与图6.9中的结果相似。

从结果中可以发现, 大约有 80% 的文件能够成功进行变更语义影响分析。无法完成分析的原因分为几种, 可能是因为该文件定义了某个抽象类或接口类而没有具体实现, 也可能是因为该文件没有成功完成程序间语法差异性分析过程, 还可能是因为该文件编译失败而没能提供相应的源代码文件, 或者可能是因为该文件确实超过了 jpf-regression 工具的分析能力等等。

### 6.3.3 冲突判定模块

在冲突判定模块中, 应用本文提出的冲突检测算法后, 再通过影响追踪系统进行辅助人工分析, 可以得到如表6.11的结果。结果表明该冲突判定模块能够找

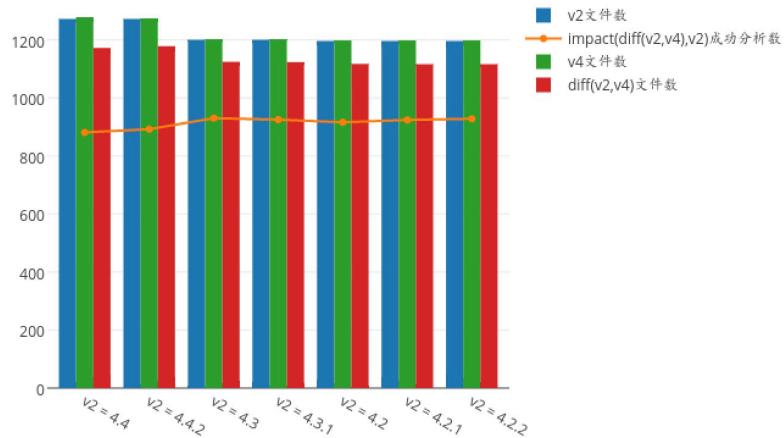


图 6.10 影响分析模块

到可能发生了语义冲突的代码位置，经过人工分析，发现这些找到的代码位置确实发生了语义冲突。

表 6.11 冲突判定结果

| 代码               | $v_2$ | 冲突文件数 | 影响域重叠 |
|------------------|-------|-------|-------|
| Eclipse JDT Core | 4.4   | 2     | 2     |
| Eclipse JDT Core | 4.4.2 | 3     | 3     |
| Eclipse JDT Core | 4.3   | 3     | 3     |
| Eclipse JDT Core | 4.3.1 | 3     | 3     |
| Eclipse JDT Core | 4.2   | 4     | 4     |
| Eclipse JDT Core | 4.2.1 | 3     | 3     |
| Eclipse JDT Core | 4.2.2 | 4     | 4     |

然而，在不使用过滤算法的情况下，实验结果如表6.12所示。将表6.11和表6.12中的数据进行对比，如图6.11所示，冲突判定模块确实能够找到补丁间的语义冲突。但是在不使用过滤算法时误报的影响域重叠数量陡增，这主要是差异性分析模块的误差所导致的。可见，程序间语法差异性分析算法的正确性对于后续分析过程而言极为重要，前置模块的误差会在后续模块的结果中得到显著体现。

表 6.12 冲突判定结果

| 代码               | $v_2$ | 冲突文件数 | 影响域重叠 |
|------------------|-------|-------|-------|
| Eclipse JDT Core | 4.4   | 2     | 59    |
| Eclipse JDT Core | 4.4.2 | 3     | 64    |
| Eclipse JDT Core | 4.3   | 3     | 69    |
| Eclipse JDT Core | 4.3.1 | 3     | 66    |
| Eclipse JDT Core | 4.2   | 4     | 64    |
| Eclipse JDT Core | 4.2.1 | 3     | 63    |
| Eclipse JDT Core | 4.2.2 | 4     | 65    |

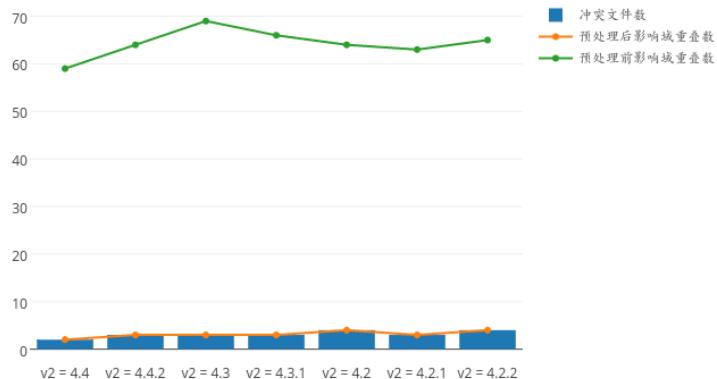


图 6.11 冲突判定模块

由于运用了过滤算法，实验中找到的冲突结果都是由于针对不同语句的变更影响到了相同的语句而被挖掘出来的。事实上，如果存在修改相同语句的多个变更，其所导致的冲突在版本合并的过程中就会暴露出来。

由于前置的影响域分析模块中的变更语义影响分析算法所计算得到的是限制在变更所处方法内部的，以基本块为单位的变更影响域，因而可能会导致对冲突的漏报，这可能是造成找到的语义冲突数量较少的主要原因。

从以上结果中可以发现，本文所实现的兼容性检测工具确实能够成功分析工业界的实际项目代码，如 Eclipse JDT Core（仅有少部分代码无法得出分析结果），也确实能够找到补丁间的语义冲突，这些语义冲突分散并深藏在上千个代码文件中，很难被直接发现。目前检测工具能够找到的语义冲突数量较少，这种情况可能是由于确实只有少量语义冲突存在，或者可能是由于工具的精度不够高、分析

的变更影响范围不够广等因素而导致的，具体是哪种情况还有待以后做进一步的分析，虽然不排除可能出现语义冲突数量确实很少的情况，但本文认为第二种情况的可能性更高。

综上所述，本文中所实现的兼容性检测工具对于工业界的实际项目而言是可用的、正确的，然而其实用性还有待进一步的提高。

#### 6.4 本章小结

本章中主要介绍了实验的设计、实验案例的选取，并给出了实验的结果和相关分析。章节6.1中主要介绍了实验的设计，包括了设计的目的以及实验的过程。章节6.2中主要介绍了实验案例的选取过程。章节6.3中主要介绍了实验的结果和对结果的分析。

## 第 7 章 结论

### 7.1 工作总结

首先，本文对软件补丁兼容性检测问题和其应用场景进行了介绍。本文主要研究在软件演化的背景下，不同软件版本的补丁能否共享，即如何确定补丁对其他软件版本的兼容性。

其次，本文对该问题进行了深入的分析和讨论，发现该问题可以归结为多次变更间的冲突检测问题，并提出了一套软件补丁兼容性检测方法用于解决该问题，该检测方法包括以下部分：

- 软件变更影响域分析，该分析过程主要用于获取变更的语义影响域（即变更影响域），分为两个子过程：
  - 程序间语法差异性分析：该分析过程用于获取不同版本代码间的变更集合，该变更集合描述了程序间的语法结构上的差异性。
  - 变更语义影响分析：根据差异性分析过程找到的变更集合，分析其变更影响域并输出，该变更影响域描述了软件系统中直接或间接受到该变更集影响的语法结构集合。
- 软件变更冲突检测：该冲突检测方法根据得到的不同变更影响域，找到变更影响域之间的重叠，该重叠位置即变更间可能存在冲突的位置。最后冲突的确定需要人工分析的辅助。

其中变更影响域分析的两个子过程可以自由替换为符合要求的相应算法实现，以提高检测方法的实用性。而冲突检测过程提出了一种较简单的自动冲突分析算法并进行了实现，更精确的分析结果目前需要人工分析的辅助，为此检测方法中需要变更语义影响分析过程提供相应的支持，以便追溯影响的来源。

最后，本文对该套检测方法给出了具体的工具设计和实现方案，并在 Eclipse JDT Core 项目上进行了测试，结果表明该工具是确实可用且有效的。它确实能够挖掘出补丁间的语义冲突并向用户进行报告。

综上所述，本文的主要贡献在于对软件补丁兼容性检测的问题进行了分析，并提出了一套补丁兼容性检测方法和其相应的兼容性检测工具实现。根据该工具在中型项目 Eclipse JDT Core 的八个不同版本上的实验结果，该检测工具对于工业界实际项目来说是可用的、正确性。

## 7.2 未来工作

对于本文中所提到的软件补丁兼容性检测方法和其工具实现而言，可能的未来工作方向包括：

- 更换兼容性检测工具中影响域分析模块所使用的分析工具，并进行进一步的实验，讨论兼容性检测工具的精度与其所采用的具体工具之间的关系。
- 将兼容性检测工具对更多的工业界实际项目进行实验，进一步探讨其实用性。

## 参考文献

- [1] Lehnert S. A review of software change impact analysis. Ilmenau University of Technology, Tech. Rep, 2011..
- [2] Pigoski T M. Practical software maintenance: best practices for managing your software investment. John Wiley & Sons, Inc., 1996.
- [3] Le W, Pattison S D. Patch verification via multiversion interprocedural control flow graphs. Proceedings of the 36th International Conference on Software Engineering. ACM, 2014. 1047–1058.
- [4] Hunt J W, MacIlroy M. An algorithm for differential file comparison. Bell Laboratories, 1976.
- [5] Buckley J, Mens T, Zenger M, et al. Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice, 2005, 17(5):309–332.
- [6] Wilkerson J W. A software change impact analysis taxonomy. Software Maintenance (ICSM), 2012 28th IEEE International Conference on. IEEE, 2012. 625–628.
- [7] Tao Y, Dang Y, Xie T, et al. How do software engineers understand code changes?: an exploratory study in industry. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, 2012. 51.
- [8] Kim M, Notkin D, Grossman D, et al. Identifying and summarizing systematic code changes via rule inference. Software Engineering, IEEE Transactions on, 2013, 39(1):45–62.
- [9] Lahiri S K, Vaswani K, Hoare C A. Differential static analysis: opportunities, applications, and challenges. Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM, 2010. 201–204.
- [10] Winstead J, Evans D. Towards differential program analysis. Proc. ICSE 2003 Workshop on Dynamic Analysis, 2003. 37–40.
- [11] Fluri B, Wursch M, PInzger M, et al. Change distilling: Tree differencing for fine-grained source code change extraction. Software Engineering, IEEE Transactions on, 2007, 33(11):725–743.
- [12] Gall H C, Fluri B, Pinzger M. Change analysis with evolizer and changedistiller. IEEE Software, 2009, 26(1):26–33.
- [13] Li B, Sun X, Leung H, et al. A survey of code-based change impact analysis techniques. Software Testing, Verification and Reliability, 2013, 23(8):613–646.
- [14] Acharya M, Robinson B. Practical change impact analysis based on static program slicing for industrial software systems. Proceedings of the 33rd international conference on software engineering. ACM, 2011. 746–755.
- [15] De Lucia A, Fasano F, Oliveto R. Traceability management for impact analysis. Frontiers of Software Maintenance, 2008. FoSM 2008. IEEE, 2008. 21–30.
- [16] Law J, Rothermel G. Incremental dynamic impact analysis for evolving software systems. Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on. IEEE, 2003. 430–441.

- [17] Bohner S A. Software change impact analysis. 1996..
- [18] Bohner S A. Software change impacts—an evolving perspective. *Software Maintenance, 2002. Proceedings. International Conference on. IEEE*, 2002. 263–272.
- [19] Biggerstaff T J, Mitbander B G, Webster D. The concept assignment problem in program understanding. *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, 1993. 482–498.
- [20] Sun X, Li B, Li B, et al. A comparative study of static cia techniques. *Proceedings of the Fourth Asia-Pacific Symposium on Internetworks*. ACM, 2012. 23.
- [21] Kagdi H, Collard M L, Maletic J I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 2007, 19(2):77–131.
- [22] Law J, Rothermel G. Whole program path-based dynamic impact analysis. *Software Engineering, 2003. Proceedings. 25th International Conference on. IEEE*, 2003. 308–318.
- [23] Ren X, Shah F, Tip F, et al. Chianti: a tool for change impact analysis of java programs. *ACM Sigplan Notices*, volume 39. ACM, 2004. 432–448.
- [24] Buckner J, Buchta J, Petrenko M, et al. Jripples: A tool for program comprehension during incremental change. *IWPC*, volume 5, 2005. 149–152.
- [25] Rajlich V, Gosavi P. Incremental change in object-oriented programming. *Software, IEEE*, 2004, 21(4):62–69.
- [26] Zimmermann T, Zeller A, Weissgerber P, et al. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 2005, 31(6):429–445.
- [27] Person S, Yang G, Rungta N, et al. Directed incremental symbolic execution. *ACM SIGPLAN Notices*, volume 46. ACM, 2011. 504–515.
- [28] Rungta N, Person S, Branchaud J. A change impact analysis to characterize evolving program behaviors. *Software Maintenance (ICSM), 2012 28th IEEE International Conference on. IEEE*, 2012. 109–118.
- [29] Yang G, Person S, Rungta N, et al. Directed incremental symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2014, 24(1):3.
- [30] Havelund K, Pressburger T. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2000, 2(4):366–381.
- [31] Petrenko M, Rajlich V. Variable granularity for improving precision of impact analysis. *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on. IEEE*, 2009. 10–19.

## 致 谢

首先要感谢贺飞老师对我的悉心指导，他对我的毕设工作和毕业论文的写作给出了许多宝贵的意见，并且不厌其烦的帮助我对论文的内容编排和组织形式进行修改。我被其认真负责的作风受到了深深的感染。

其次要感谢实验室的同学和师兄等的帮助，如郭心睿、周旻、刘盛鹏等，他们在写论文的过程中提供了很多帮助，让我铭感于心。

最后要感谢 ThuThesis，帮助我顺利完成了本文的写作。

## 声 明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名： \_\_\_\_\_ 日 期： \_\_\_\_\_

## 个人简历、在学期间发表的学术论文与研究成果

### 个人简历

1990 年 04 月 03 日出生于四川省绵阳市。

2008 年 9 月考入北京理工大学软件学院软件工程专业，2012 年 7 月本科毕业并获得工学学士学位。

2012 年 9 月进入清华大学软件学院攻读工程硕士学位至今。