

# 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



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 : 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

众所周知，软件维护是软件开发周期中耗时最长，也是开销最大的过程。软件维护过程中，经常伴随着软件版本不断演进的过程。在实际应用中，补丁程序是最常见的用于软件版本演进的一类程序，它能够实现软件功能增强、漏洞修复等。

补丁程序的局限性在于，它一般都是针对于某个专门版本的代码而设计，通常无法正确的在其他版本代码上使用。然而，在实际开发过程中工程师经常能够遇到这样的问题，如果对其他版本代码重新开发专门的补丁程序，其资源消耗较多。

为此，本文中提出了一套基于语义影响域分析的软件补丁兼容性检测方案，并且实现了相应的工具。语义影响域分析可以挖掘变更对于软件代码的语义影响，根据该分析过程中得到的语义影响域，我们能够进行补丁间的语义冲突检测。通过在 Eclipse JDT Core 上的实验，我们发现该工具确实能够发现软件补丁的兼容性问题，本文中所提出的解决方案是可用的、可靠的。

**关键词：**软件维护；语义影响域；补丁；兼容性检测

## Abstract

As well known, software maintenance is the longest and most cost activity in the software developing cycle. In software maintenance, the software system sometimes got evolved. In practical application, patch is the most common program which used for software evolving, it can enhance its functionality, fix its bug and so on.

The limitation of patch program is that it's usually designed for a specific version of software so that it cannot be applied to another version directly. However, it is a common question happened during the software development and developing another patch program for this specific version seems too costly.

Under the above circumstance, we propose a solution called software patch compatibility checking based on semantic impacted area analysis for this problem and develop a corresponding tool. The semantic impacted area analysis can mine semantic impact introduced by software change from the code. And using its result we could check if there is a semantic conflict issue between the patches. After experimenting on the Eclipse JDT Core project, the tool is proven to have the ability to find out compatibility issues between the patches and this solution is applicable and sound.

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

## 目 录

<b>第 1 章 绪论 .....</b>	<b>1</b>
1.1 研究背景和意义 .....	1
1.2 本文所要解决的问题与主要工作 .....	2
1.2.1 应用场景 .....	2
1.2.2 问题解读 .....	2
1.2.3 主要工作 .....	3
1.3 本文组织结构 .....	3
<b>第 2 章 相关工作 .....</b>	<b>5</b>
2.1 程序间差异分析 .....	5
2.2 程序变更影响分析 .....	5
2.3 相关工具 .....	8
2.3.1 git .....	8
2.3.2 Beyond Compare .....	9
2.3.3 jpf-regression .....	9
2.3.4 ASTro .....	10
<b>第 3 章 分析方法 .....</b>	<b>11</b>
3.1 问题定义 .....	11
3.1.1 补丁应用 .....	11
3.1.2 补丁冲突 .....	12
3.2 问题分析 .....	14
3.2.1 补丁应用 .....	14
3.2.2 补丁冲突 .....	15
3.3 解决方案 .....	18
3.3.1 整体架构 .....	18
3.3.2 补丁应用 .....	19
3.3.3 语义影响域分析 .....	20
3.3.3.1 程序差异性分析 .....	20
3.3.3.2 变更影响分析 .....	21
3.3.4 冲突分析 .....	22
3.4 本章小结 .....	23

## 目 录

---

<b>第 4 章 工具设计与实现 .....</b>	<b>24</b>
4.1 整体架构 .....	24
4.2 版本迁移模块 .....	26
4.2.1 设计 .....	26
4.2.2 实现 .....	28
4.3 影响域分析模块 .....	30
4.3.1 差异性分析模块 .....	30
4.3.1.1 设计 .....	30
4.3.1.2 实现 .....	32
4.3.2 影响分析模块 .....	36
4.3.2.1 设计 .....	36
4.3.2.2 实现 .....	38
4.4 冲突判定模块 .....	45
4.4.1 设计 .....	45
4.4.2 实现 .....	47
4.5 本章小结 .....	50
<b>第 5 章 实验结果与分析 .....</b>	<b>51</b>
5.1 实验设计 .....	51
5.2 实验案例 .....	52
5.3 实验结果与分析 .....	53
5.3.1 版本迁移模块 .....	53
5.3.2 影响域分析模块 .....	55
5.3.2.1 差异性分析模块 .....	55
5.3.2.2 影响分析模块 .....	56
5.3.3 冲突判定模块 .....	57
5.4 本章小结 .....	58
<b>第 6 章 结论 .....</b>	<b>60</b>
6.1 工作总结 .....	60
6.2 未来工作 .....	60
<b>参考文献 .....</b>	<b>62</b>
<b>致 谢 .....</b>	<b>64</b>
<b>声 明 .....</b>	<b>65</b>
<b>个人简历、在学期间发表的学术论文与研究成果 .....</b>	<b>66</b>

## 第1章 绪论

### 1.1 研究背景和意义

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，同时也需要修复在实际运行中暴露出来的问题。

在软件演进的过程中，软件系统可能会由于各种各样的原因而发生更新行为，这是软件维护周期中的常见活动<sup>[2]</sup>。为了进行软件版本演进，现代软件工程中提出了许多解决方案。

补丁（Patch）就是这样其中一类可以用于完成修补程序漏洞、增强软件功能、改善程序性能等任务的程序。补丁在工业界中得到了广泛的实际应用，是软件维护过程的重要组成部分<sup>[3]</sup>。

补丁一般是通过 Diff 工具而产生的文本数据<sup>[4]</sup>，用于表示以行为单位的程序间差异性。虽然得到了广泛应用，但补丁程序仍然具有一定的局限性。

首先，Diff 工具实质上是属于程序差异性分析工具中的一种，它进行单纯的文本差异性比较，并给出行级别的代码差异性信息。然而这样产生出来的差异性信息（Patch），只包含了文本差异，而失去了语法、语义等其他信息。

其次，它一般只针对某个专门软件版本而开发，对于软件演进过程中获得的新版本而言，我们无法确定补丁程序是否也能适用于新版本的程序。而且补丁程序的开发一般都具有特定的目的，例如功能升级、漏洞修补等，新版本的程序很可能仍然需要应用补丁程序来完善自身。因而在实际的软件维护过程中工程师往往不得不重新去开发针对该新版本的特定补丁程序，对人力成本和时间资源造成了许多浪费。

究其缘由，这主要是由于补丁程序通常会引入软件变更（Software Change）<sup>[5]</sup>，使得软件源代码在应用这些变更之后可以实现功能修复、版本升级等目的。使用 Diff 工具所产生的补丁文件一般会以行为单位引入变更。

然而，软件变更的影响不仅仅局限于被修改的某行，事实上，由于软件代码的耦合性，单行变更就足以将变更带来的影响广泛地传播到软件系统的其他部分中去<sup>[6,7]</sup>。因此，软件变更不仅会使软件系统发生语法结构的变化，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到该变量的条件语句。

因此，如何确定补丁程序对于软件其他版本的适用性就成为了一个较复杂并

且具有实际意义的问题。这个问题主要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。实际上，也就是变更所影响到的程序结构之间的冲突，而这种冲突既可能是语法结构上的，也可能是语义层面上的。

可见，软件补丁的兼容性检测是一个较为复杂并且具有实际意义的问题，就我们所知，目前尚无这方面的相关工作。鉴于该问题具有实际意义且普遍存在，是工业界中常见的问题，本文将尝试去解决之。

## 1.2 本文所要解决的问题与主要工作

如前所述，软件系统总是处于不断演进的过程中的。补丁 (Patch) 是一种常见的软件系统版本演进的方法，常用于修补漏洞和缺陷、改进性能、升级等。在实际应用中，补丁往往针对某一个特定版本的代码而设计，然而现代软件系统往往更新换代较快，因而面临着将补丁应用于其他版本的代码时可能失效的问题。

为此，我们考虑这样一个问题，针对某软件系统  $s$ ，假设已有版本  $v_1$  和  $v_2$ ，其中补丁  $p_1$  是针对版本  $v_1$  而设计，应用后能使其演进到版本  $v_3$ ，问  $p_1$  是否能够成功应用于版本  $v_2$ ，并保证不会发生冲突？

### 1.2.1 应用场景

本文中的主要问题在于如何判定一个针对给定程序版本的补丁，它是否能够应用于其他版本，并且与其兼容？为了使读者对这一问题更加清晰，我们可以考虑这样一个应用场景：

- 某个团队正在对一软件进行开发工作，该软件已推出一个正式版  $v_1$ ，现在正在进行  $v_2$  版本的开发。
- 该团队使用版本控制系统进行版本管理，使用 Github 作为代码托管和团队协作工具。
- 该软件存在一个第三方开发的针对  $v_1$  版本的的补丁  $p$ ，该补丁可以增强该软件的功能。

我们希望知道，该补丁  $p$  是否能够应用于正式版  $v_2$ ，使得补丁  $p$  中的变更不仅能够被正确地应用，并且软件版本  $v_2$  的功能正确性不会受到影响？

### 1.2.2 问题解读

事实上，对该问题而言，其答案可以分为多个层面来回答。

从语法角度出发，则该问题主要关注的是在应用过程中是否会造成语法结构上的错误，如果能够将补丁  $p_1$  成功应用于程序版本  $v_2$ ，则认为该补丁是可以兼容于新版本  $v_2$  的。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。

从语义角度出发，则单纯的语法结构上的兼容性并不全面。事实上，某行修改可能会影响到多处其他源代码结构，从而导致程序的行为发生变化。因此，从语义层面而言，我们需要保证补丁  $p_1$  对程序版本  $v_2$  造成的语义不会影响到从版本  $v_1$  演进到  $v_2$  时所引入的语义变化。也就是说我们需要保证补丁  $p_1$  和  $p_2$  之间不会有语义上的冲突。

语法层面的回答很容易就能给出，现有的版本控制系统等工具都足以给出相应的答案。而对于语义层面来说，就目前所知，尚未有这方面的工作出现。

因此，本文将着重从语义层面上去尝试解决这个问题。详细的问题讨论可以参考章节3.1。

### 1.2.3 主要工作

目前而言，本文的主要工作包括以下几个部分：

1. 提出软件补丁兼容性检测问题。
2. 提出解决该问题的一套解决方案，包括：
  - 补丁应用：将转为某个版本代码而设计的补丁应用于其他版本代码。
  - 语义影响域分析：分析并获取软件变更对代码的影响域。
  - 冲突分析：根据得到的影响域，分析是否存在语义冲突。
3. 根据解决方案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。

可见，本文的主要贡献在于，提出了补丁兼容性检测问题的通用解决方案，并给出了相应的工具实现。并且，根据兼容性检测工具在工业界实际项目 Eclipse JDT Core 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

本文主要包括六个章节，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；第三章主要介绍了补丁兼容性分析的具体方法；第四章主要介绍了如何将各阶段的不同分析方法进行整合，

形成具体的流程；第五章介绍了实验过程和结果；第六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

### 2.1 程序间差异分析

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活动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。他们会追踪对某个特定文件的文本行的增加/删除等操作，并没有考虑代码中的结构化变更。

考虑到源代码可以抽象语法树（Abstract Syntax Tree）的形式进行表达，可以考虑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 的实体和语句中抽取信息，该算法可以获取结构化的变更信息。

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合适数量的移动操作。

### 2.2 程序变更影响分析

软件维护是软件开发周期中最复杂、高成本和劳动密集的活动。软件产品天然的需要跟随系统需求的变更而进行适应和变化，以满足用户的需求。软件变更更是软件维护中的基础组件。变更可能从新的需求、缺陷修复、变更请求等而来，当变更应用于软件时，他们不可避免的会带来一些副作用，也可能会与原软件的其

他部分产生不协调。

而 CIA 正是这样一类用于确定变更对于软件其他部分的影响的技术集合<sup>[13]</sup>，它在软件开发、维护和回归测试等过程中都起到着重要的作用<sup>[14]</sup>。一般而言，CIA 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。

CIA 方法的分类：

- 基于追踪性的 CIA<sup>[15]</sup>，追踪两个不同抽象级别上的元素间的依赖性，目的在于链接不同类型的软件工件（需求、设计等）。
- 基于依赖关系的 CIA<sup>[16]</sup>，致力于衡量变更的潜在影响，通常尝试去分析程序的语法关系，他们表示了程序实体间的一些语义依赖关系，或者说他们在同一个抽象级别上对软件工件实施了影响分析。现在最主要的这类 CIA 主要在源代码级别上进行研究。

软件系统上的变更可能导致不可意料的副作用，CIA 的目标就在于辨认这些副作用 (side effect 或者 ripple effect)<sup>[17]</sup>，并防止之。CIA 从分析变更请求和源代码开始，以便获得变更集合 (change set)。最后实际获得的影响集合叫做 EIS (estimated impact set)，而真实的集合叫做 AIS (actual impact set)。

整个软件变更影响的分析过程可以大致分为以下流程，更详细的流程可以参考图 2.3<sup>[15,18]</sup>。

1. 确定变更集合，需要针对变更请求进行分析，这步一般叫做特征定位 (feature location)，用于确认源代码中实现了对应功能的起始位置<sup>[19]</sup>。
2. 衡量由变更集合引入的影响。这一步是目前大部分 CIA 技术的着重点。若按分析技术的类型划分，则主要的两类包括静态分析和动态分析。
  - 静态分析：历史分析、文本分析、结构分析<sup>[20,21]</sup>。静态分析通过分析程序的语法、语义或者历史依赖实现，通常会产生很多错报 (False Positive)。
    - 结构分析着重于分析程序的结构依赖性，构建依赖关系图
    - 文本分析根据注释和标识符提取出概念依赖性
    - 历史分析从多个软件演进版本中挖掘出信息

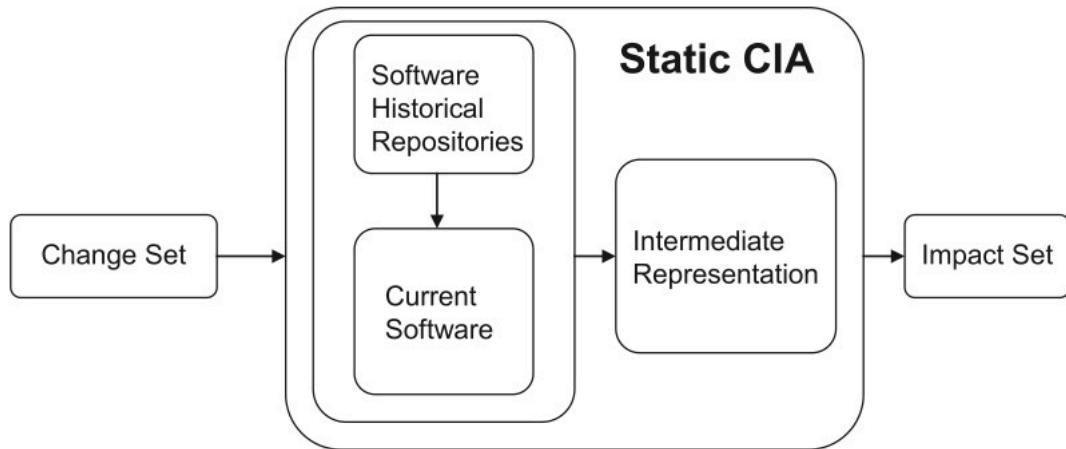


图 2.1 静态分析过程

- 动态分析：在线分析和离线分析。考虑特定的输入，并且依赖于程序运行时收集到的信息进行分析（如运行时的追踪信息、覆盖信息等）<sup>[22]</sup>。其得到的影响集合往往比静态分析精度更高，但其开销也相应更大，而且通常包括错报（False Negative）。
  - 在线 CIA 使用程序运行时收集的信息实施所有的分析

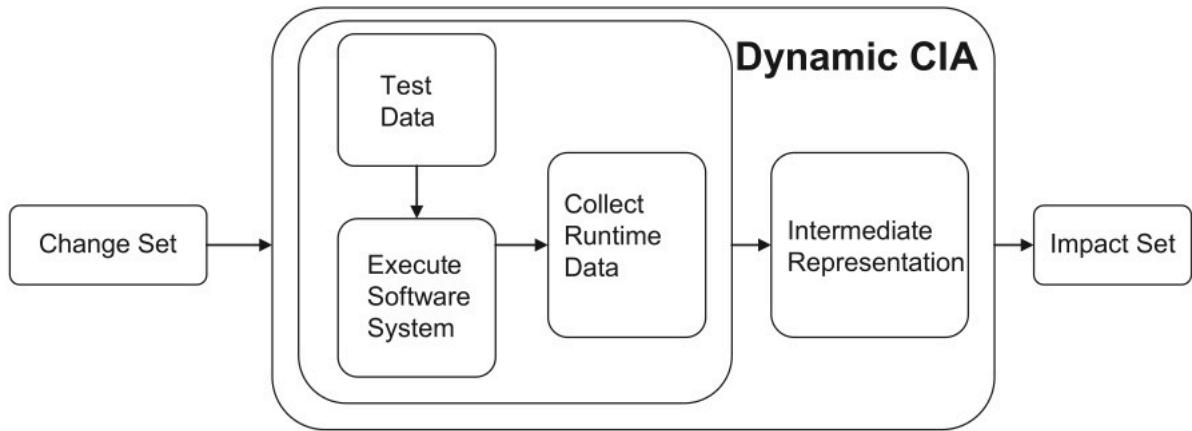


图 2.2 动态分析过程

近年来，涌现出一些以 CIA 技术作为支撑而实现的工具，它们一般在实现 CIA 技术之后，利用得到的影响集合进行后续分析，帮助进行软件维护和演进。下面对它们进行简要的介绍：

1. Chianti：应用于 Java 程序，Eclipse 插件<sup>[23]</sup>。

它可以：

- (a) 使用回归测试来分析变更之前的程序功能是否能正常使用。
- (b) 若测试失败，利用 Chianti 进行变更影响分析，将软件变更拆分成若干

原子变更，并分析他们之间的依赖关系，结合原程序生成中间表示形式，分析可能是哪些部分影响到了测试用例的运行。

2. JRipples: 应用于 Java 程序，Eclipse 插件<sup>[24,25]</sup>。

它利用依赖关系图，自动标注可能受到被变更的类所影响的其他类，并向用户展示其变更的影响传播路径。可利用人工标注来修正结果。

3. ROSE: 应用于 Java CVS 工程，Eclipse 插件<sup>[26]</sup>。

它可以挖掘软件代码仓库，当用户对代码变更时提示用户可能相关的变更（类似于“变更了这个函数的人通常还变更了另一个函数”）。

4. jpf-regression: 应用于 Java 程序，Eclipse 插件或命令行工具<sup>[27]</sup>。

它可以利用程序切片技术进行变更影响分析，利用得到的影响集合来驱动符号化执行，获取被改变部分代码的行为。

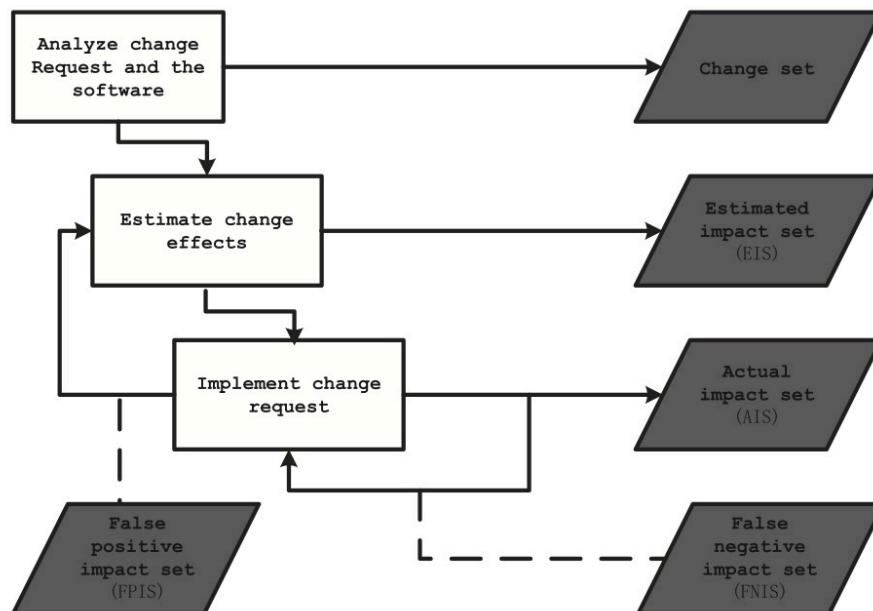


图 2.3 CIA 过程

## 2.3 相关工具

本节将主要介绍本文中采用到的相关工具。

### 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 是分布式的版本库，

每个本地的 git 工作目录都包含了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。

本文中主要考虑以 git 作为版本管理系统的应用场景，类似于 GitHub，假定为项目开发了 new version 和 patch version 两个不同的分支，并使用 git 的分支合并策略实现补丁的版本迁移过程。

### 2.3.2 Beyond Compare

Beyond Compare 是一款内容比较工具，可以用于文件、目录、压缩包的比较，横跨 Windows、Mac OS X、Linux 三大操作系统，可用作版本控制系统的文本比较和合并工具，例如 git。

本文中主要采用其作为 git 的文本比较和合并工具，用于解决补丁版本迁移时的冲突问题。

### 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。CIA 的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 CIA 来确定程序的哪些部分需要进行再分析。而由于软件开发过程的演进特性，并且软件系统的大小和复杂性越来越高，刺激了人们对于有效的变更影响分析的需求。<sup>[28]</sup>

一般而言，“单行变更”就足以引发广泛而未知的影响，因而变更影响分析在软件演进和维护过程中扮演着重要的作用。

目前，大部分的自动分析工具都将变更的影响以程序句法结构的形式表现出来，例如函数和程序语句。基于依赖的分析方法则通过分析程序组件间的内部关系来衡量变更的影响。这类技术能够以程序位置信息来描述变更的影响，然而却缺失了与受影响位置相关的程序运行路径的信息，而这类信息往往对于程序行为的验证、调试等起到很大的作用，而且能够将关注的范围缩小，只关注受到影响的程序行为集合。

(Directed Incremental Symbolic Execution ) DiSE<sup>[27,29]</sup> 能够结合静态分析的效率和符号化执行的精度，分析方法内部的变更影响，并描述程序变更对其运行行为的影响。

DiSE 的静态分析部分采用程序切片技术来衡量变更对代码中其他位置的影响，其生成的影响集合可以用于引导符号化执行去分析受到影响的程序行为，并生成对应的路径条件 (path condition)，这些路径条件用于描述受到影响的程序行为。这些路径条件之后可以利用 SMT 等技术进行求解，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进行后续分析过程的基础，它的变更影响分析过程主要特点在于：

1. 粒度：语句级别。即具体的影响分析过程中，以程序语句为基本单位进行影响集合的计算。
2. 影响范围：方法内部。即以方法的作用域作为单次影响集合计的限制范围，最后得到变更对所处的方法内部其他语句所造成的影响。也就是说不计算变更对于其他方法的影响，即不考虑对方法内部函数调用的影响。
3. 影响来源：主要从控制流和数据流两个层面考虑变更所造成的影响，实际上采用了语句间的控制依赖和数据依赖关系进行计算。

DiSE 中，受影响的集合主要分为两类：

- 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）。

为了计算得到这两类影响集合，DiSE 中一共使用了四条规则来迭代计算，从最初的变更集合出发，不断应用规则向外扩展，最后计算得到闭包，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。

前两条公式表明，如果条件语句和赋值语句控制依赖于变更后的 CFG 中的节点，那么他们就应当被加进来。

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。

jpf-regression 是 DiSE 的工具实现，它是基于 Java Path Finder 框架<sup>[?]</sup>而实现的工具，可作为 Eclipse 的插件使用，支持 Java 语言。

### 2.3.4 ASTro

本文中所采用的 AST 比较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，它也是 jpf-regression 中使用的前置工具，用于对比两个不同版本的源代码，并获取其抽象语法树上的差异，以 XML 文件的格式进行输出。该工具支持 Java 语言。

## 第3章 分析方法

### 3.1 问题定义

第一章中对本文所要解决的问题进行了简要介绍。下面对该问题进行正式的定义。

**问题 3.1：** 针对某软件系统  $s$ ，假设已有版本  $v_1$  和  $v_2$ ，其中补丁  $p_1$  是针对版本  $v_1$  而设计，应用后能使其演进到版本  $v_3$ ，问  $p_1$  是否能够应用于版本  $v_2$ ，并且应用之后是否会发生冲突？

考虑到补丁间不兼容的实质是因为在应用时和应用后出现了错误，可以具体将错误分为两类：

**定义 3.1：** 兼容性错误——将补丁  $p_i$  和补丁  $p_j$  先后应用于同一版本代码  $v_k$  时和之后所出现的程序错误。主要分为两类：

- 语法错误：即在应用一补丁  $p_i$  之后，再次应用另一补丁  $p_j$  时出现语法结构上的错误。易由编译器发现。
- 语义错误：将补丁  $p_i$  中的变更和补丁  $p_j$  中的变更应用于源代码之后，由于语义上的变更互相冲突而导致出现了功能上的错误，较难发现。

可以发现，该问题的核心包括两点，即：

- 如何将一个专为版本  $v_1$  设计的补丁  $p_1$  成功应用于版本  $v_2$  上。即如何消去补丁应用过程中的语法错误。
- 如何检测补丁在应用于新版本之后是否会与该版本代码产生冲突？即如何检测补丁应用之后的语义错误。

下面分别对这两点进行分析和阐述。

#### 3.1.1 补丁应用

实际上，补丁程序是专门为某一版本的软件系统所设计，因而往往不能直接应用到其他版本的代码上，而且强行应用往往会出现很多错误。常见的问题可能包括：

1. 补丁程序中所提及的行不在原位置。
2. 补丁程序中所要删除的行其实已被删除，或者所要添加的行其实已被添加。
3. 补丁程序中所提及的文件未找到。

由于有这些可能的问题存在，导致我们无法直接将补丁程序应用到其他版本的补丁上。因此，我们考虑如何去解决问题3.1的第一点：

由于补丁  $p$  针对版本  $v_1$  设计，当尝试将其应用到版本  $v_2$  上时，可能会出现程序语法结构上的错误。如何将  $p$  成功应用到版本  $v_2$  上，使得该过程不会出现引入程序语法错误？

为了更清楚的描述，我们可以将其进一步形式化定义如下：

**定义 3.2:**  $Code$ ——指源代码。

**定义 3.3:**  $Patch$ ——指补丁，即变更的集合。

**定义 3.4:**  $patch : Patch \times Code \mapsto Code$ 。 $patch(p_i, v_k) = v_m, v_k, v_m \subset Code, i, k, m \in \mathbb{N}$ 。该函数用于表示补丁应用的过程。

**定义 3.5:**  $Pat : Code \mapsto \{Patch\}$ 。用于表示代码到其对应的补丁集合的映射。

**定义 3.6:**  $compile : Code \mapsto Boolean$ 。 $compile$  函数用于表示对源代码的编译过程，如果成功编译则表示源代码中没有语法错误。

**定义 3.7:** 给定  $p_i, v_k$ ，问  $p_i \notin Pat(v_k)$  时，能否使得  $compile(patch(p_i, v_k)) == True$ ？其中  $p_i \subset Patch, v_k \subset Code, i, k \in \mathbb{N}$ 。

可见，要解决问题3.1的第一点，其核心在于如何给出合适的  $patch(p_i, v_k)$  函数。

### 3.1.2 补丁冲突

就如何检测补丁在应用于新版本之后是否会与该版本代码产生冲突而言，直接去寻找冲突的存在是一件让人困惑的事情，什么是冲突？为什么会发生冲突？怎么进行检测？

因此本文考虑从另一个角度来看待这个问题。

实际上，考虑到从版本  $v_1$  到版本  $v_2$  的演进过程同样可以使用补丁来完成。那么可以考虑将这个演进过程表述成为合适的变更，得到另外一个补丁。

**定义 3.8:**  $diff : Code \times Code \mapsto Patch$ 。其中  $p = diff(v_i, v_j) \iff patch(p, v_i) = v_j, i, j \in \mathbb{N}, p \subset Pat(v_i)$ 。该函数用于求解两个不同版本的程序  $v_i$  和  $v_j$  之间的差异性，其结果即为补丁  $p$ 。

从问题3.1中可以发现， $p_1 = \text{diff}(v_1, v_3)$ 。如果同样给定 $p_2 = \text{diff}(v_1, v_2)$ ，并且已经解决了问题3.1的第一点，那么，对于问题3.1的第二点而言，如何检测补丁在应用于新版本之后是否会与该版本代码产生冲突就可以规约如下：

给定针对某同一版本代码 $v_k$ 的补丁 $p_i$ 和 $p_j$ ，问分别应用于 $v_k$ 之后，这两个补丁对于该版本代码的语义影响是否会产生冲突？

这样一来，问题的实质就清楚了。显然，该问题是由于两个补丁同时对同一版本代码造成了语义影响而造成的。那么为了回答这个问题，我们需要明确的知道到底什么是补丁对代码的语义影响，以及什么是冲突？

本文首先讨论什么是补丁对代码的语义影响。

**定义 3.9：**语义影响（Semantic Impact）——假如某个代码结构对其他代码结构存在依赖关系，那么就说这两者间存在语义影响。其中代码结构指程序的语法结构，包括类、方法、语句等不同的类型。

事实上，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。可见，这里所谓的语义影响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。

下面给出语义影响的形式化定义。

**定义 3.10：**  $\text{Structure} \rightarrow \text{Class} \mid \text{Method} \mid \text{Statement} \mid \dots$ 。Structure 用于表示程序代码结构，可以任意替换为对应的子类型。

**定义 3.11：**  $\text{Struct} : \text{Code} \mapsto \{\text{Structure}\}$ 。Struct 函数定义为一个从源代码到该源代码的所有代码结构集合的映射。

**定义 3.12：**  $im : \text{Structure} \mapsto \text{Structure}$ 。影响关系定义为一个从代码结构到代码结构的映射关系。

**定义 3.13：**  $depend : \text{Structure} \mapsto \text{Structure}$ 。依赖关系定义为一个从代码结构到代码结构的映射关系。

**定义 3.14：**  $\forall \text{structure}_i, \text{structure}_j \subset \text{Struct}(v_k), im(\text{structure}_i) = \text{structure}_j \iff depend(\text{structure}_i) = \text{structure}_j$ ，其中 $v_k \subset \text{Code}, i, j, k \in \mathbb{N}$ 。

**定义 3.15：**语义冲突（Conflict）——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。其中两条变更互斥指二者的语义互相影响。

下面给出其形式化的定义。

**定义 3.16:**  $Change$ ——指变更。 $Patch = \{Change\}$ 。

**定义 3.17:**  $change : Structure \mapsto Change$ 。用于表示代码结构和其所对应的变更的映射。

**定义 3.18:**  $conflict : Patch \times Patch \mapsto Boolean$ 。如果  $\exists structure_i, structure_j, structure_k \subset v_t$ , 使得  $im(structure_i) = structure_k \wedge im(structure_j) = structure_k$ , 那么  $conflict(p_m, p_n) = True$ 。其中  $change(structure_i) \subset p_m, change(structure_j) \subset p_n, p_m, p_n \subset Pat(v_t), v_t \subset Code, i, j, k, m, n, t \in \mathbb{N}$ 。用于判断两个补丁是否发生语义冲突的函数。

在有了语义影响和冲突的定义之后, 问题3.1的第二点就可以形式化的描述如下:

给定  $v_k \subset Code, p_i, p_j \subset Pat(v_k), i, j, k \in \mathbb{N}$ , 问是否有  $conflict(p_i, p_j) == True$ ?  
因此, 问题3.1可以形式化的定义如问题3.2所示。

**问题 3.2:** 给定  $p_i, v_k, v_m$ , 问当  $p_i \notin Pat(v_k)$  时, 如果能够使得  $compile(patch(p_i, v_k)) == True$ , 并且  $\exists p_j = diff(v_k, v_m)$ , 那么是否有  $conflict(p_i, p_j) == True$ ? 其中  $p_i \subset Patch, p_j \subset Pat(v_k), v_k, v_m \subset Code, i, j, k, m \in \mathbb{N}$ 。

## 3.2 问题分析

### 3.2.1 补丁应用

如前所述, 为了解决问题3.1的第一点, 主要需要提供合适的  $patch(p_i, v_k)$  函数。实际上可以采用版本控制系统的版本合并功能来作为该函数的具体实现。

版本控制系统中经常需要将其他分支中的版本合并到当前分支中, 为了解决不同版本之间可能存在的语法冲突, 通常会采用三路归并算法 (3-way merge) 来对两个不同版本的代码进行合并, 合并时以二者共同的祖先版本作为依据进行。

而我们所面临的问题也是类似的, 由于  $patch(p_1, v_1) = v_3$ , 为了将  $p_1$  应用到版本  $v_2$  上获得版本  $v_4$ , 我们可以采用类似的想法, 将版本  $v_3$  和版本  $v_2$  进行合并即可。合并后我们就等于同时拥有了  $p_1 = diff(v_1, v_3)$  和  $p_2 = diff(v_1, v_2)$  这两个补丁中的变更。

整个过程可以形式化定义如下。实际上也就是说我们用  $merge(v_i, v_j)$  函数实现了  $patch(p_i, v_k)$  的功能。

**定义 3.19:**  $\text{merge} : \text{Code} \times \text{Code} \mapsto \text{Code}$ 。 $v_s = \text{patch}(p_i, v_l) = \text{merge}(v_m, v_l)$ , 使得  $\forall c_a \subset p_i, \forall c_b \subset p_j, c_a, c_b \subset p_k$ , 即  $p_k = p_i \cup p_j$ 。其中  $v_m = \text{patch}(p_i, v_n), p_j = \text{diff}(v_n, v_l), p_k = \text{diff}(v_l, v_s), p_i \notin \text{Pat}(v_l), p_i, p_j \subset \text{Pat}(v_n), p_k \subset \text{Pat}(v_l), v_l, v_m, v_n, v_s \subset \text{Code}, a, b, i, j, k, l, m, n, s \in \mathbb{N}$ 。

这实际上也是现在软件开发过程中的常见手段。例如使用 git 作为版本控制系统时, 为了修复某个功能性的 bug, 我们可以新建一个分支 FixBug, 然后切换到该分支进行漏洞修复, 等到修补完毕后, 再将该分支合并到主分支 Master 中即可。

该过程如图3.1所示。

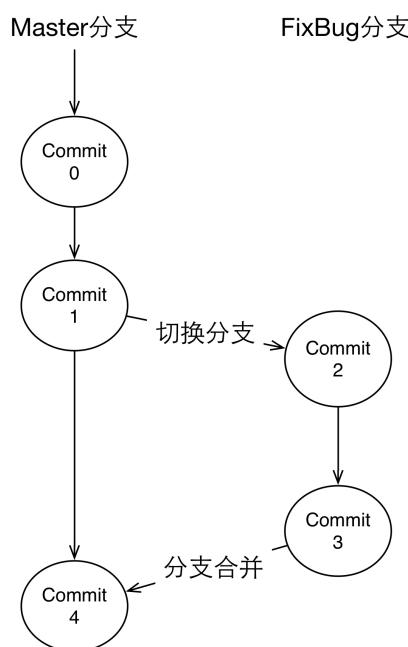


图 3.1 git 分支切换与合并

### 3.2.2 补丁冲突

可以发现, 为了解决问题3.1的第二点, 我们主要需要从代码中挖掘中两类信息:

1. 代码的变更集合, 即两个版本之间代码的差异性。用于进行问题规约。
2. 变更的影响集合, 即变更集合对代码中其他哪些部分会造成影响。用于确定变更造成的语义影响。

在有了这两类集合之后, 我们就可以界定出软件变更在语义层面上对代码的语义影响域, 我们可以将整个过程称为语义影响域分析 (Semantic Impacted Area Analysis)。

下面首先给出语义影响域的概念。

**定义 3.20:** 程序结构——指程序的基础语法结构，包括类、方法、基本块、语句等不同级别。

**定义 3.21:** 语义影响域 (Semantic Impacted Area)——在程序的某个限定范围内，直接或间接受到变更影响的程序结构集合。

可见，语义影响域描述了变更对于程序中其他部分的语义影响。为了更清楚的理解，下面给出影响域的形式化定义。

**定义 3.22:**  $impact : Patch \times Code \mapsto \{Structure\}$ 。  
 $impact(p_m, v_k) = \{\forall structure_j \subset Struct(v_k) \mid structure_j = im(structure_i) \wedge (change(structure_i) \subset p_m \vee structure_i \subset impact(p_m, v_k)), structure_i, structure_j \subset Struct(v_k), v_k \subset Code, p_m \subset Pat(v_k), i, j, k, m \in \mathbb{N}\}$ 。该函数用于描述补丁  $p_m$  对于其所应用的某个版本代码  $v_k$  的影响域。其中可以将影响关系  $im$  限定在不同  $Structure$  类型的范围内。

为了挖掘出这两类信息，可以将语义影响域分析划分为两个子过程：

1. 程序间差异性分析：获取两个软件版本间的差异性，获得结构化的软件变更信息。该分析过程即为  $diff(v_1, v_2)$  函数的具体实现。
2. 变更影响分析：分析软件变更对软件其他部分是否存在影响，并找到受影响的集合。该分析过程即为  $impact(p_1, v_1)$  函数的具体实现。

我们可以将整个语义影响域分析过程定义为如下的函数：

**定义 3.23:**  $ia : Code \times Code \mapsto Structure$ 。  
 $s = ia(v_i, v_j) = impact(diff(v_i, v_j), v_i), i, j \in \mathbb{N}$ 。该分析过程先将版本演进的过程转化为补丁，再去计算补丁对程序代码的语义影响域。

通过语义影响域分析，我们就可以从代码中挖掘出所需要的变更影响信息，从而可以进行后续的分析工作。

事实上，可以将程序中受变更影响的部分划分为不同的粒度，从而获得不同程度的影响<sup>[30]</sup>，我们考虑对面向对象的编程方法中的影响元素级别进行划分：

1. 类：探讨变更对于其他类和对象的影响。对于面向对象的程序设计方法而言，这是最高级别的粒度。
2. 方法：探讨变更对于其他方法的影响。
3. 基本块：探讨变更对于其他基本块的影响。
4. 语句：探讨变更对于其他语句的影响。

而所谓的影响范围也需要界定其粒度，不同层级的粒度显然会对语义影响域分析的精度产生影响。

1. 类间：考虑变更的影响可能延伸到其他类、对象。
2. 方法间：考虑变更的影响可能延伸到其他方法内部。
3. 方法内部：考虑变更的影响只在本方法的内部延伸。

在实际情况中，不同级别的影响范围均可分别采用不同的影响元素级别。

在有了软件变更的语义影响域之后，我们可以通过简单地比较两个补丁对代码的影响域是否存在重叠来判定其是否发生了语义冲突。因此可以给出更详细的语义冲突的定义。

**定义 3.24：**语义冲突 (Semantic Conflict)——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。其中两条变更互斥指二者对同一行代码进行了修改，或者他们所修改了不同行代码，但其影响传播到了某行相同的代码。

上述定义是对语义冲突的非形式化的描述，为了更清楚的理解，可以给出如下的形式化定义。

**定义 3.25：**如果  $\exists c_m \subset p_i, \exists c_n \subset p_j$ ，其中  $c_m = change(structure_m), c_n = change(structure_n)$ ,  $structure_m \subset Struct(v_k), structure_n \subset Struct(v_k), v_k \subset Code$ , 并且  $i, j, k, m, n \in \mathbb{N}$ 。那么有  $conflict(p_i, p_j) == true \iff (structure_m == structure_n) \vee (impact(p_i, v_k) \cap impact(p_j, v_k) \neq \emptyset)$ ，其中  $p_i, p_j \subset Pat(v_k)$ 。

可见，如果两个补丁对代码的影响域不存在重叠，那么他们两者之间自然是兼容的，因为他们不仅本身的变更互不干涉，并且他们所影响到的程序结构也互不干涉。在这样的情况下，补丁是可以成功应用到其他版本上的代码，并且可以完成补丁本身的目标的。

如果影响域发生了重叠，那么我们就可以认为补丁之间是不相容的，因为补丁所作的变更会对相同的程序结构产生影响。

实际上这样的影响是否能够兼容还需要人工的判定，因为在不同的情况下，他们可能是兼容的，也可能是不兼容的。而我们无法从代码中获取到足够的信息来进行这样的判定，需要外界对于变更的期望信息作为辅助。

我们可以将这个过程定义为如下类型的函数，并称之为冲突分析。可见，我们实际上是采用了  $isCompatible(s_i, s_j)$  函数来作为  $conflict(p_i, p_j)$  的具体实现。

**定义 3.26：** $isCompatible(s_i, s_j) : \{Structure\} \times \{Structure\} \mapsto Boolean$ 。其中  $conflict(p_i, p_j) = isCompatible(impact(p_i, v_i), impact(p_j, v_j)), i, j \in \mathbb{N}$ 。

显然，可以发现，对于问题3.1的第二点，其核心在于：

- 如何规约该问题并计算影响域？即  $diff(v_i, v_j)$  和即  $impact(p_i, v_k)$  函数的具体实现。也就是整个语义影响域分析的过程。
- 如何判定发生了冲突？即  $conflict(p_i, p_j)$  函数的实现。也就是整个冲突分析的过程。

### 3.3 解决方案

考虑前两节中所提到的应用场景和对问题的分析，我们可以给出一个通用的解决方案来解决整个兼容性问题的判定。

#### 3.3.1 整体架构

根据章节3.2中的问题分析，实际上整个问题3.1的解决过程可以参考图3.2。其输入输出的描述参考表3.1。

可见，整个解决方案的实现过程包括三个过程：

- 补丁应用过程。即  $merge$  函数的实现过程。
- 语义影响域分析过程。即  $diff$  函数和  $impact$  函数的实现过程。
- 冲突分析过程。即  $conflict$  函数的实现过程。

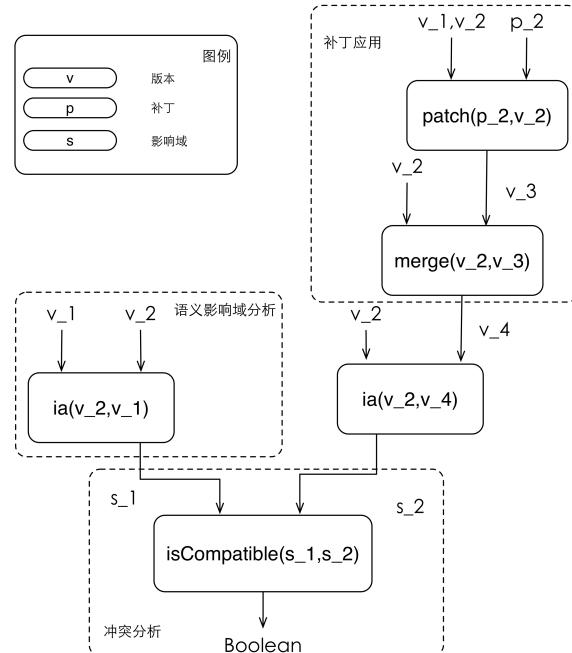


图 3.2 解决方案

表 3.1 输入输出对照表

输入输出	描述
$v_1$	旧版本源代码，待检测的对象
$v_2$	新版本源代码
$v_3$	将补丁 $p_2$ 应用于版本 $v_2$ 后的源代码
$v_4$	将补丁 $p_2$ 中的变更应用于版本 $v_1$ 后的源代码
$p_2$	适用于 $v_2$ 的补丁，待检测的对象
$s_1$	$diff(v_2, v_1)$ 对版本 $v_2$ 的影响域
$s_2$	$diff(v_2, v_4)$ 对版本 $v_2$ 的影响域
Boolean	是否冲突

### 3.3.2 补丁应用

如章节3.2.1中所述，我们可以采用常见的版本控制工具提供的三路归并算法进行补丁的版本迁移工作。具体来讲，其流程可以简述如下：

1. 将补丁  $p_1$  应用到版本  $v_1$ ，获得旧版本应用补丁后的代码，即其版本为  $v_3 = patch(p_1, v_1)$ 。
2. 采用三路归并算法实现  $v_4 = merge(v_2, v_3)$  过程，获得新版本应用补丁后的代码  $v_4$ 。
3. 解决分支合并中可能出现的冲突问题。

其中，通过三路归并算法进行的合并工作中可能会出现冲突，这说明版本  $v_2$  和  $v_3$  之间存在语法冲突，即这两个版本在语法层面上不兼容。我们可以通过人工修改的方式进行修复，实现语法层面上的兼容性。

在解决冲突的过程中可以采用第三方的合并工具，利用现成工具的高效算法快速解决冲突。

该补丁应用的过程主要用于解决补丁的版本适应性问题，整个过程可以用算法1描述如下。

**Algorithm 1** 补丁应用**Require:**  $v_1, v_2, p_1$ **Ensure:**  $v_4$ 

```
1:  $v_3 \leftarrow patch(p_1, v_1)$ 
2:  $v_4 \leftarrow merge(v_2, v_3)$ 
3:  $v_4 \leftarrow resolve(v_4)$ 
4: return  $v_4$ 
5:
6: function merge(new, old_patched)
7:     return 3-way-merge(old, new, old_patched)
8: end function
9:
10: function resolve(version)
11:    return mergetool(version)
12: end function
```

---

### 3.3.3 语义影响域分析

如章节3.2.2中所述，我们需要进行语义影响域分析来获取两个补丁的影响域，用于进行冲突分析使用。

实际上，语义影响域分析主要分为两个分析过程，即程序间差异分析和变更影响分析，通过这两个分析的协作来完成整个语义影响域的分析。

#### 3.3.3.1 程序差异性分析

程序差异性分析是 *diff* 函数的实现。它主要用于分析两个不同版本的程序间的差异性，其结果即我们所需要的程序变更集合。近年来程序差异性分析方面有不少工作，实现了一些较为成熟的比较工具，我们可以使用这些工具来实现该分析过程。

在本文的组合架构中，程序差异性分析的主要任务是接受两个不同版本的源代码，并返回代码间的结构化差异信息。结构化的差异信息可以视为补丁的一种，只不过它具有比常见的采用 Unix *diff* 工具生成的 *.patch* 类型的补丁文件更丰富的信息，能够描述以程序语法结构的形式对软件变更进行描述。

该分析过程应该满足如下需要：

- 输入为两个不同版本的代码。

- 输出为源代码间的软件变更集合。
- 每条变更描述语句或基本块级别的变更。
- 每条变更描述新旧程序结构的相关信息和其关联关系。
- 每条变更描述了其所属的作用域。

选择这样的分析结果类型是为了后续分析过程的方便，因为变更影响分析需要我们提供软件变更集合作为输入，而 *.patch* 类型的补丁文件只描述文本行的变更，不包含语法信息，我们无法从中提取出所需的语法层面的变更信息。

一种比较好的选择是采用 AST 差异性分析，因为抽象语法树中包括了足够多的语法结构信息。

### 3.3.3.2 变更影响分析

程序变更影响分析是 *impact* 函数的具体实现。主要用于获取变更集合对其他程序结构的影响域，该集合所包含的程序结构直接或间接地受到变更集合中的元素影响。近年来这方面比较成熟的工作也有不少，因而可以直接选择合适的变更影响分析算法作为该过分析过程的具体实现。

在本文的组合架构中，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影响集合，这也就是我们所需要的变更集合的语义影响域。变更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结构是否存在影响，并进行闭包计算即可。

本文对于该分析过程的要求如下：

- 接受两个不同版本代码间的变更集合作为输入。
- 计算得到该变更集合所对应的影响集合。
- 计算过程中可以指定受影响的范围和元素类型。
- 影响集合中的元素按照其不同类型进行分类。
- 具有影响追踪系统，将计算影响域的过程进行记录。方便后续的冲突分析过程进行回溯。

其中影响追踪系统可以定义成如下类型的函数。该函数接受一个影响域分析函数  $ia(v_i, v_j)$ ，并返回对应的依赖关系集合。

**定义 3.27:**  $impact\_track(ia(v_i, v_j)) : (Code \times Code \mapsto Structure) \mapsto depend$

在完成了上述两项子分析过程后，我们就完成了整个语义影响域分析过程，并获得了两个不同版本代码的变更集合和对应的其语义影响范围。接下来就可以进行具体的兼容性分析工作。

整个语义影响域分析过程可以用算法2描述。

**Algorithm 2** 语义影响域分析算法

---

**Require:**  $v_i, v_j$ **Ensure:**  $s$ 

```
1: function ia( $v_i, v_j$ )
2:    $p \leftarrow diff(v_i, v_j)$ 
3:    $s \leftarrow impact(p, v_i)$ 
4:   return  $s$ 
5: end function
```

---

### 3.3.4 冲突分析

冲突分析即为 *conflict* 函数的具体实现。我们将对比两个版本代码的影响域，确定他们是否重叠，以此为依据判断其兼容性。

理论上来讲，这种简单比对即可发现两个版本间的兼容性问题，因为一旦发生重叠，那么重叠的代码部分显然是会发生兼容性问题的。然而在实际情况中，受限于工具的精度，我们往往不能达到理论上的准确度，而可能会发生误报（False Negative）等情况。

显然，如果重叠不存在，则兼容性是可以得到满足的。而对于如何界定重叠部分的兼容性，则需要人工分析的介入，因为这部分代码的兼容性与补丁的功能目标密切相关，而我们无法从代码中获取到这种信息，因而只有依靠外界来提供，并以此为依据进行详细分析过程，界定这部分代码是否真的存在冲突。

在进行人工分析的过程中，我们不仅需要知道哪部分代码出现了重叠现象，而且还需要知道是哪些变更影响到了这部分代码，因而我们需要引入影响追踪系统来记录变更影响分析过程的轨迹。影响追踪系统可以记录下变更影响分析过程中的每一步，从而获取到程序结构间的影响关系链，事后通过回溯即可追踪到具体的软件变更可见，本过程中主要需要影响追踪系统的回溯模块的支持。

该分析过程可以参考算法3。

**Algorithm 3** 冲突分析**Require:**  $s_1 = ia(v_2, v_1), s_2 = ia(v_2, v_4)$ 1:        $t_1 = impact\_track(ia(v_2, v_1)), t_2 = impact\_track(ia(v_2, v_4))$ **Ensure:**  $isCompatible(diff(v_2, v_1), diff(v_2, v_4))$ 2:   **if**  $s_1 = \emptyset \vee s_2 = \emptyset$  **then**3:        $result \leftarrow True$ 4:   **else**5:        $s \leftarrow s_1 \cap s_2$ 6:       **if**  $s = \emptyset$  **then**7:            $result \leftarrow True$ 8:       **else**9:            $result \leftarrow Manual\_analysis(s_1, s_2, t_1, t_2)$ 10:      **end if**11:     **end if** **return**  $result$ 

### 3.4 本章小结

本章主要介绍了软件补丁兼容性检测所关注的主要问题以及如何解决该问题。

章节3.1介绍了问题定义。

章节3.2中对问题进行了深入分析和讨论，提出了解决该问题所需要的分析工作。

章节3.3介绍了如何将前一节中提到的分析方法进行整合，形成一套通用解决方案，同时还对其中每个分析方法所需要遵守的约定和算法进行了描述。

## 第4章 工具设计与实现

本章中将主要介绍如何设计与实现补丁兼容性分析工具。章节3.3给出了一套通用解决方案，本章中主要考虑如何对该方案进行具体的实现并工具化。

### 4.1 整体架构

根据章节3.3.1中的描述，工具可以设计成如图4.1所示的组合形式。

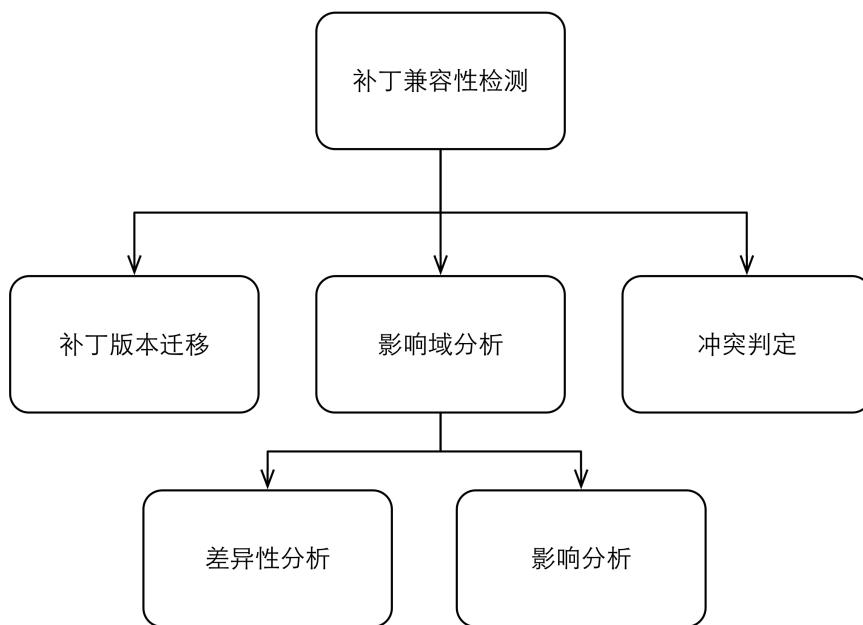


图 4.1 整体架构

可见，兼容性检测工具可以拆分成三个模块：

- 补丁版本迁移模块：将为某个版本而设计的专门补丁应用于其他版本代码。
- 影响域分析模块：分析并获取变更的语义影响域。
  - 差异性分析模块：分析代码间的差异性，产生对应的变更集合。
  - 影响分析模块：根据变更集合，分析变更对于代码的语义影响，产生对应的影响域。
- 冲突判定模块：根据影响域分析模块所得到的影响域，判定两个补丁之间是否产生冲突。

整个工具的运作流程可以参考图4.2。其输入输出可以参考表4.1。

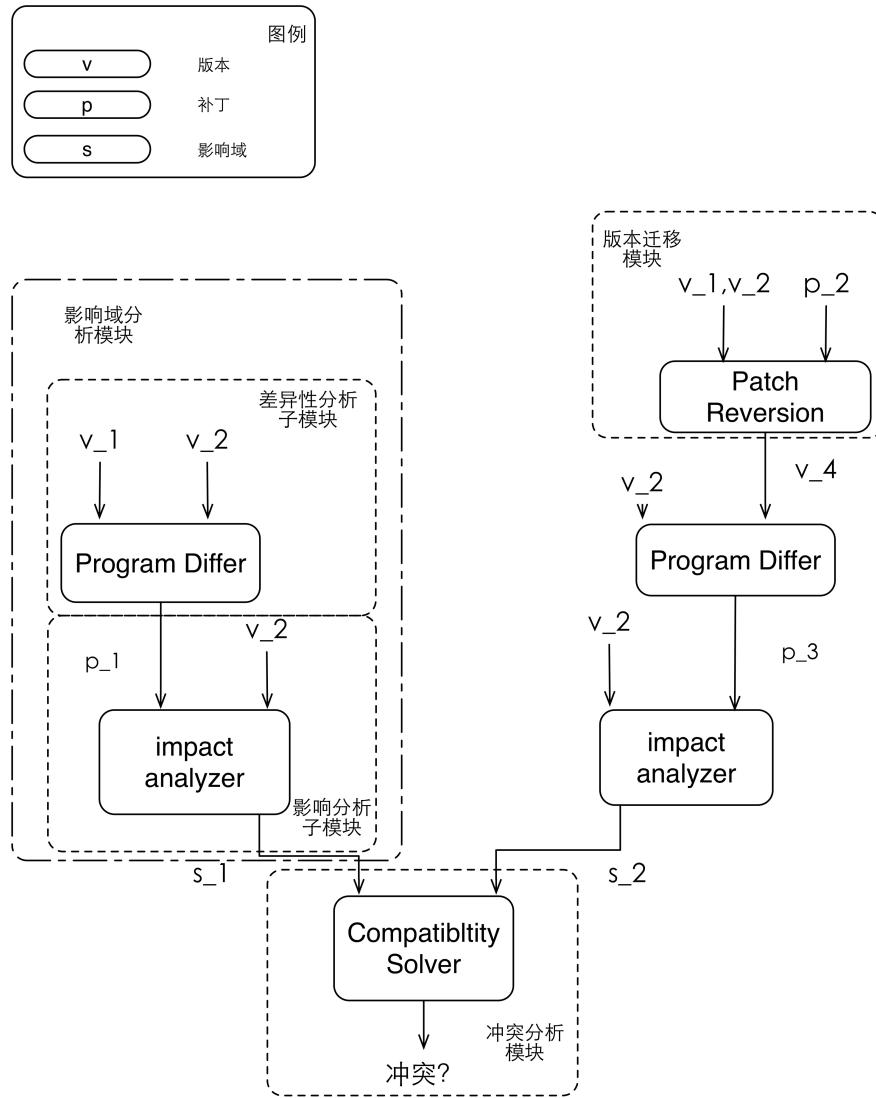


表 4.1 输入输出对照表

输入输出	描述
$v_1$	旧版本源代码，待检测的对象
$v_2$	新版本源代码
$v_4$	将补丁 $p_2$ 中的变更应用于版本 $v_1$ 后的源代码
$p_2$	适用于 $v_2$ 的补丁，待检测的对象
$p_1$	实质上是版本 $v_1$ 和 $v_2$ 之间的变更集合
$p_3$	实质上是版本 $v_4$ 和 $v_2$ 之间的变更集合
$s_1$	变更集合 $p_1$ 对版本 $v_2$ 的影响域
$s_2$	变更集合 $p_3$ 对版本 $v_2$ 的影响域

其整体流程可以简述如下：

1. 采用版本控制系统进行代码版本管理，并进行补丁版本迁移，得到应用于新版本的补丁后代码版本  $v_4$ 。
2. 根据得到的三个版本代码  $v_1$ 、 $v_2$ 、 $v_4$ ，分别分析其程序间差异性，生成对应的程序变更集合。
3. 根据得到的程序变更集合，进行不同版本间的变更影响分析，生成语义影响域。
4. 根据得到的语义影响域，进行冲突分析。

下面分别对工具中各个模块的设计与实现进行介绍。

## 4.2 版本迁移模块

如前所述，补丁版本迁移模块主要用于解决问题3.1的第一点。该模块主要需要实现  $merge$  函数的实际功能。

我们在实现该模块时，主要采用了 git 工具和 Beyond Compare 工具来实现版本合并和冲突解决的过程。

### 4.2.1 设计

在该模块的设计中，其输入输出过程可以描述如图4.3，输入输出的具体描述参见表4.2。

显然，该模块主要完成的任务为实现  $merge$  函数，其流程为：

1. 实现  $v_4 = patch(v_2, p_1)$ ，即将补丁  $p_1$  应用于版本  $v_2$
2. 实现  $v_3 = merge(v_1, v_4)$ ，即将版本  $v_1$  和  $v_4$  进行代码合并，使得  $p_1$  中的变更应用于版本  $v_1$  上。

该模块的设计可以参考图4.4。

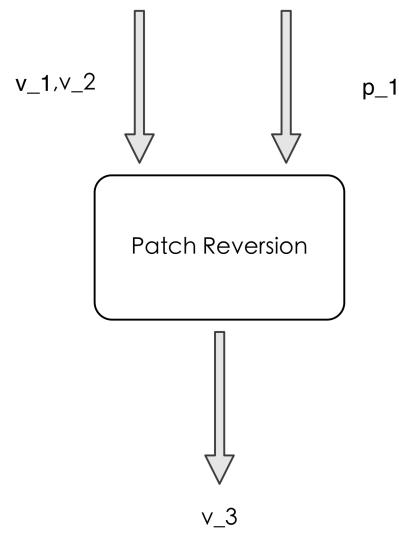


图 4.3 差异性分析模块

表 4.2 输入输出对照表

输入输出	描述
$v_1$	源代码
$v_2$	源代码
$p_1$	为版本 $v_2$ 设计的补丁
$v_3$	将补丁 $p_1$ 中的变更应用于版本 $v_1$ 后的源代码

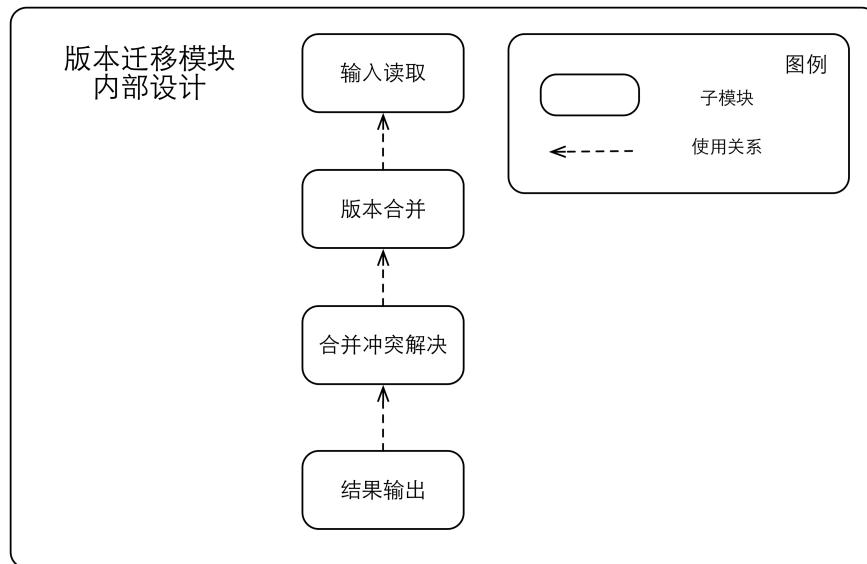


图 4.4 模块设计

### 4.2.2 实现

该模块的实现中，使用 git 工具完成了具体的 *merge* 函数功能。

因而实际上该模块的输入输出可以参考表4.3。

表 4.3 输入输出对照表

输入输出	描述	格式
输入	源代码	Java
输入	源代码	Java
输入	补丁	patch
输出	源代码	Java

下面介绍该过程的详细实现。首先介绍一些前提假设：

1. 假设 git 的工作目录如图4.5所示。其中 `git_working_dir` 为 git 目录，下属目录包括源代码仓库 `src` 和 git 目录的隐藏文件夹`.git`。另外与 git 目录平级的目录 `patch` 下面则存储了具体的补丁文件。
2. 假设补丁  $p_1 = \text{old.patch}$ ，并且采用 git diff 或其他 Unix diff 命令生成。

```
+/git_working_dir
  |+/.git
  |+/src
+/patch
  |+old.patch
```

图 4.5 git 工作目录

下面将具体介绍整个版本合并的过程。

首先在工作目录下将 git 切换到主分支 master。然后在主分支 master 中提交代码版本  $v_1$ 。

```
1 git checkout master
2 git add .
3 git commit -a -m "old.version.committed"
```

然后新建并切换到分支 new 中，并提交代码版本  $v_2$ 。

```
1 git checkout -b new
2 git add .
3 git commit -a -m "new.version.committed"
```

再次从主分支新建并切换到分支 patch，然后将补丁  $p_1$  应用到版本  $v_1$ ，获得旧版本应用补丁后的代码，其版本为  $v_3$ ，并提交代码版本  $v_3$ 。此时我们所应用的补丁  $p_1$  是专为版本  $v_1$  设计，所以应用时不会出现问题。

```
1 git checkout master
2 git checkout -b patch
3 git apply ..patch/old.patch
4 git add .
5 git commit -a -m "patched version from old version committed"
```

然后再切换回分支 new，将分支 patch 合并入分支 new，获得新版本应用补丁后的代码，其版本为  $v_4$ ，并使用 Beyond Compare 工具解决可能出现的冲突问题。将冲突解决完毕后，再提交版本  $v_4$ 。

```
1 git checkout new
2 git merge patch
3 git mergetool
4 git commit -a -m "patched version from new version committed"
```

如果确实有冲突，那么 git mergetool 命令会调用第三方的可视化合并工具并引导你去解决冲突。这里我们采用的合并工具即为 Beyond Compare，它会展开一个可视化的界面，并给出冲突位置的提示，方便进行人工选择、合并。

整个过程可以参考图4.6。

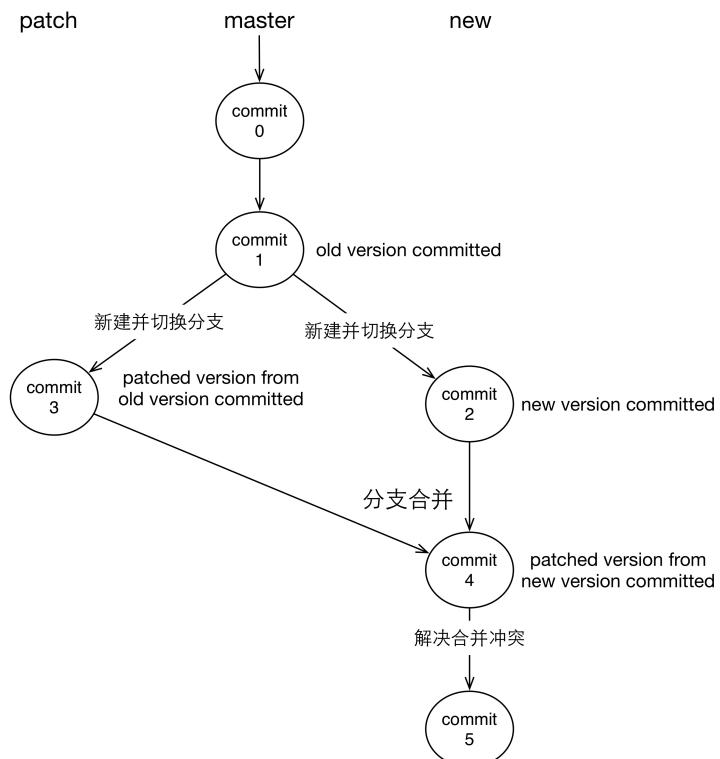


图 4.6 git 版本合并

## 4.3 影响域分析模块

如前所述，影响域分析模块主要用于生成变更的影响域。该模块需要对  $ia(v_i, v_j)$  函数进行实际的实现。

因此该模块中首先需要定义好合适的影响范围和影响元素的级别。在实际情况中，我们主要采用了较为简单的分析过程，即将影响范围限制在方法内部，但是将受影响元素设置为程序语句级别，以保证精度。

在实现该模块的过程中，由于我们可以采用现有工具来完成具体的程序间差异分析和变更影响分析过程，因而这两个子过程的分析算法可以直接使用已有的成熟算法。我们可以将主要精力放在如何整合两个子过程的，并将其工具转化为适用于本问题的具体情况的工作上。

在具体的实现过程中，我们发现一些主要需要解决的问题包括：

1. 修正 ASTro 的分析结果，以提高精确度。
2. 改进 jpf-regression 工具，包括：
  - (a) 修复工具中自带的 Bug。
  - (b) 修改工具的分析过程。
  - (c) 增加影响追踪系统。
  - (d) 增加错误记录系统。
3. 实现实验过程的批量化和自动化。

下面将分别介绍差异性分析子模块和影响分析子模块的设计与实现过程。

### 4.3.1 差异性分析模块

本文中主要采用 jpf-regression 工具自带的前置工具 ASTro 来实现差异性分析模块的功能。该子模块实现了 *diff* 函数的实际功能，接受两个不同版本的 Java 源代码文件作为输入，以 XML 格式输出程序间的变更集合。

#### 4.3.1.1 设计

在该模块的设计中，其输入输出过程可以描述如图4.7，输入输出的具体描述参见表4.4。

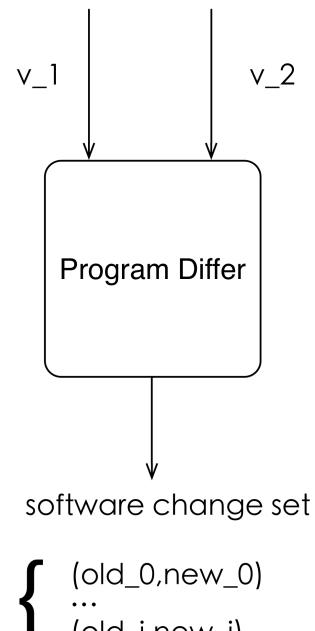


图 4.7 差异性分析模块

图中所描述的模块输出——软件变更集合，设计为如下的格式，即变更前后的代码结构所组成的二元组集合：

**定义 4.1：**  $change\_set = \{(old_i, new_i) \mid old_i \subset Structure, new_i \subset Structure, i \in \mathbb{N}\}$

由于该模块需要调用其他工具来完成具体的差异性分析过程，其输出应作为影响分析模块的输入，可见该模块的核心任务包括：

- 差异性分析
- 输入
- 输出

因此该模块的内部设计可以参考图4.8。

该模块的流程也就可以设计成如下的形式：

1. 读取输入
2. 进行差异性分析
3. 输出分析结果

表 4.4 输入输出对照表

输入输出	描述
$v_1$	源代码
$v_2$	源代码
$change\_set$	变更集合

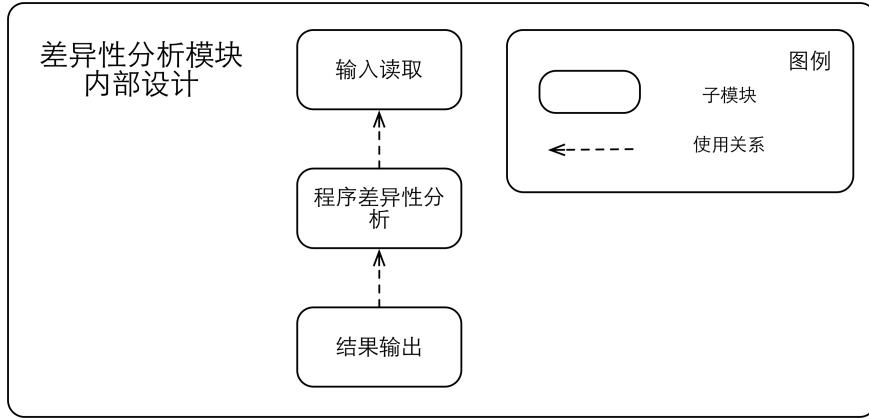


图 4.8 模块设计

#### 4.3.1.2 实现

在该模块的实现过程中，采用了 ASTro 工具来完成具体的程序间差异分析过程。

因而实际上该模块的输入输出可以参考表4.5。

表 4.5 输入输出对照表

输入输出	描述	格式
输入	源代码	Java
输入	源代码	Java
输出	影响分析模块配置文件	JPF
输出	变更集合	XML

ASTro 支持对 Java 代码的比对。它会比对两个文件的抽象语法树，并从中抽取出对应的不同之处，形成语法结构上的差异性，并输出为 XML 格式的文件以供后续分析过程使用。

该工具输出的 XML 文件将源代码按照抽象语法树的格式进行输出，其节点级别为基本块，并提供了丰富的差异性信息，如两个版本代码的对应节点是否发

生了变更等。

利用这些输出信息，我们可以从中提取出程序的变更集合，从而进行后续的变更影响分析。

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。

受限于 ASTro 工具的具体实现，其输出结果的存在一定的问题，主要包括：

1. 对某些代码文件无法完成差异性分析。
2. 对某些代码文件输出结果不准确，存在过高估计（over-estimate）的问题。

对于第一个问题，由于无法知道该工具的源代码，我们无法解决，不过这只是极少数现象。

对于第二个问题，我们分析其结果可以发现，其结果中存在误报的情况，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。同样由于无法知道该工具的源代码，我们无法从算法的角度进行修改，不过对于这样的情况，我们可以对其输出结果进行一定的预处理，将这些误报的情况进行过滤，保留一个真变更子集合即可。

预处理算法可以用伪代码4进行描述。

---

**Algorithm 4** 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:**  $\text{filter}(c_1, c_2)$

```
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:   $c_2 \leftarrow c_2.delete(del_2)$ 
```

---

我们可以归纳证明这种预处理操作的正确性。由于变更对于代码的影响是链式的，对于某次变更影响分析的结果集合  $s_{i,j} = ia(v_i, v_j)$  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  $s_{i,j}$  中被删除，但此时  $e_k$  本身即为伪影响，集合  $s_{i,j}$  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影响分析中导致  $e_k$  被添加到集合  $s_{i,j}$  中，因而也不会使集合  $s_{i,j}$  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  $s_{i,j}$  中被删除，但此时  $e_k$  本身即为伪影响，集合  $s_{i,j}$  的正确性会得到提高。

可见，我们的预处理操作是正确的，它不会导致结果集合  $s = ia(v_i, v_j)$  的正确性降低。

在实现该模块的时候，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，从而是实现对整个软件系统的所有代码进行批量化处理。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数据即可。在这部分工作中，脚本代码主要完成了以下任务：

- 实验数据定位，包括 Java 源代码和编译后的 Class 文件等。
- 根据代码的存放路径，计算其对应 Class 文件的位置。
- 获取代码文件名，以确定本次分析的对象。
- 实验数据的依赖 JAR 包定位。
- 创建输出文件目录。
- 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。
- 调用 ASTro 进行单次分析。

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4.6。

<sup>1</sup> ASTDiffer 3/27/2013

<sup>2</sup> USAGE: java ASTDiffer –original <file>.java –modified <file>.java

<sup>3</sup> –dir <output folder>

<sup>4</sup> OPTIONAL: –file <fileName> –ocp <classpath> –mcp <classpath>

<sup>5</sup> –oco <outputDir> –mco <outputDir> –cs –xml

其次，我们采用了 shell 脚本完成了对后续分析过程的支持，能够自动批量化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

表 4.6 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

表 4.7 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

值对的方式定义了各项属性的值。JPF 文件的格式等来自于 Java Path Finder 框架的设计，是该框架运行所必须的配置文件。该配置文件的具体属性为自定义，可以参考表4.7所述。

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，考虑到实际使用中，我们需要将新版本  $v_2$  作为对比的基准，以获取一致的行号。因而在进行变更影响分析时，我们需要进行相应配置，使得：

- $s_1 = impact(diff(v_2, v_1), v_2)$ , 求得变更集合  $p_1 = diff(v_2, v_1)$  对版本  $v_2$  的影响范围  $s_1$ 。
- $s_2 = impact(diff(v_2, v_4), v_2)$ , 求得变更集合  $p_3 = diff(v_2, v_4)$  对版本  $v_2$  的影响范围  $s_2$ 。

实际上，也就是说：

- 补丁  $p_1 = diff(v_2, v_1)$ , 即将版本  $v_2$  视为“旧版本”，将版本  $v_1$  视为“新版本”。
- 补丁  $p_3 = diff(v_2, v_4)$ , 即将版本  $v_2$  视为“旧版本”，将新版本应用补丁后的版本  $v_4$  视为“新版本”。

在实际操作中，我们只需做这样的版本交换即可。

整个差异性分析模块的工作流程可以参考图4.20。

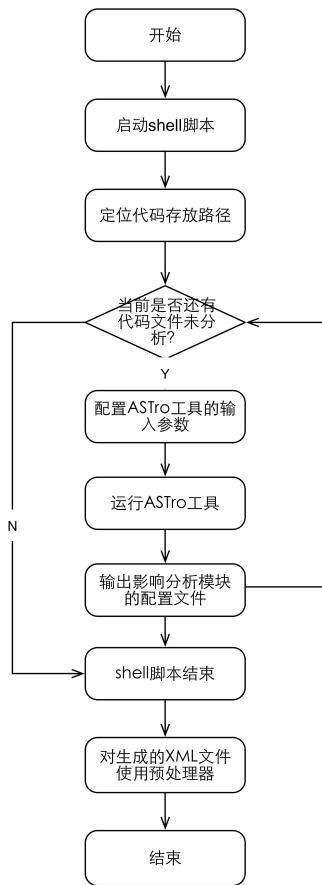


图 4.9 程序间差异性分析流程

### 4.3.2 影响分析模块

本文中主要采用 jpf-regression 工具实现影响分析模块的功能。该子模块主要实现了 *impact* 函数的实际功能，它接受 XML 格式的变更集合作为输入，并输出 Dot 格式的影响域。

#### 4.3.2.1 设计

在该模块的设计中，其输入输出过程可以描述如图4.10，输入输出的具体描述参见表4.8。

考虑到该模块需要的核心任务包括：

- 变更影响分析
- 影响追踪：记录影响分析过程的轨迹，即影响的依赖关系 *depend*。
- 输入

- 输出

因此，该模块的设计可以参考图4.11。

该模块的流程也就可以设计成如下的形式：

1. 读取输入
2. 变更影响分析，并进行影响追踪
3. 结果输出

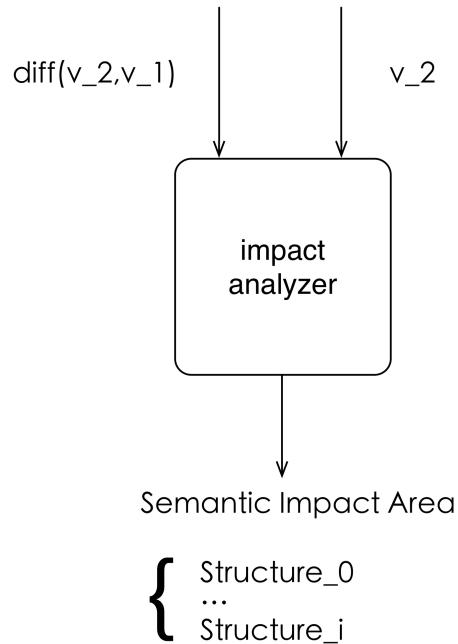


图 4.10 影响分析模块

表 4.8 输入输出对照表

输入输出	描述
$diff(v_2, v_1)$	差异性分析模块的输出，变更集合
$v_2$	源代码
Semantic Impact Area	语义影响域，即受影响代码结构的集合

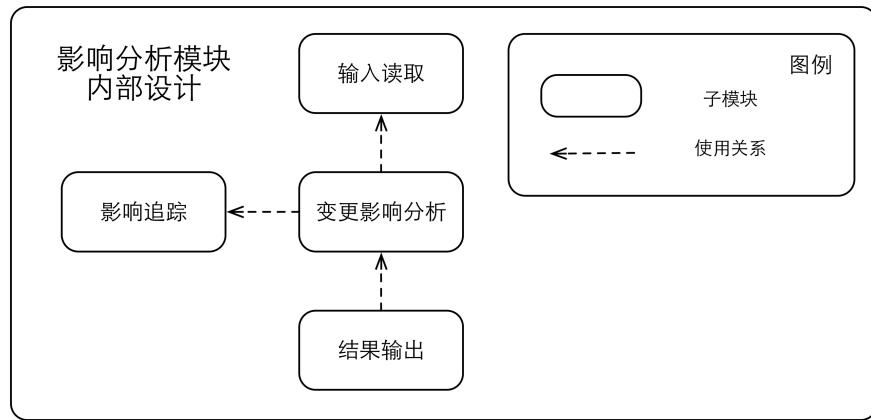


图 4.11 模块设计

#### 4.3.2.2 实现

在本模块的实现中，实际上采用了 jpf-regression 来完成具体的影响分析工作。该工具接受 Java 格式的源代码作为输入，并输出 Dot 格式的影响域信息。

可见，该模块实际上是以 Dot 格式输出的影响域信息。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 (Control Flow Graph)，影响域作为该控制流图额外承载的信息，在 CFG 上进行了标注。

影响域的相关信息主要包括两类，

- 受影响的 CFG 节点。这实际就是影响域中的元素。可见这里受影响元素的级别为基本块 (Basic Block)。
- 影响的依赖关系。这实际是记录了受影响元素与影响来源元素之间的关系。该关系的定义可以参考章节3.1中所提到的 *depend* 映射关系。

该模块的实际输入输出可以参考表4.9。

表 4.9 输入输出对照表

输入输出	描述	格式
输入	差异性分析模块的输出，变更集合	XML
输入	源代码	Java
输入	源代码	Class
输入	配置文件	JPF
输出	语义影响域 + 控制流图	Dot

jpf-regression 是 DiSE 方法在 Java Path Finder 软件框架下的具体实现，提供了方法内和方法间的程序语句级别的变更影响分析。

然而 jpf-regression 工具中变更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。因而在实践过程中，我们采用的解决办法是重用 jpf-regression 的代码，并对其加以改造，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。

下面分别进行介绍。

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，有明确的执行流程规定。然而在实际使用中，该流程约定与我们的实际情况并不合用，因而我们对此进行了一定的修正。

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，并且由于只考虑 Main 函数及其调用的函数，工具可以节约分析的开销，更快的得出结论，而忽略掉其他事实上并未在执行过程中被涉及到的函数。

然而对于我们的分析需要而言，该流程只能覆盖到部分情况，对于其他类型的软件系统而言可能并不适用，例如以 Eclipse JDT Core 项目而言，该项目主要用于为 Eclipse 软件系统的其他组件提供服务，因而在实际中以 JAR 包的形式作为库函数而存在。对于这类以库函数形式对外提供服务的源代码而言，他们并不存在入口函数，也无法预知到底会有哪些函数会被外界所调用。因而对于这类情况而言，我们需要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。

我们对于流程的修改可以参考图4.13进行对比。首先我们去掉了图4.12中所示的 JPF 框架启动流程，直接调用 jpf-regression 的核心功能。其次，如图4.13所示，我们将待计算的方法集合从 Main 函数的调用函数集合修改为文件中所有方法的集合，并在相应的影响集合计算过程中增加了影响追踪系统。

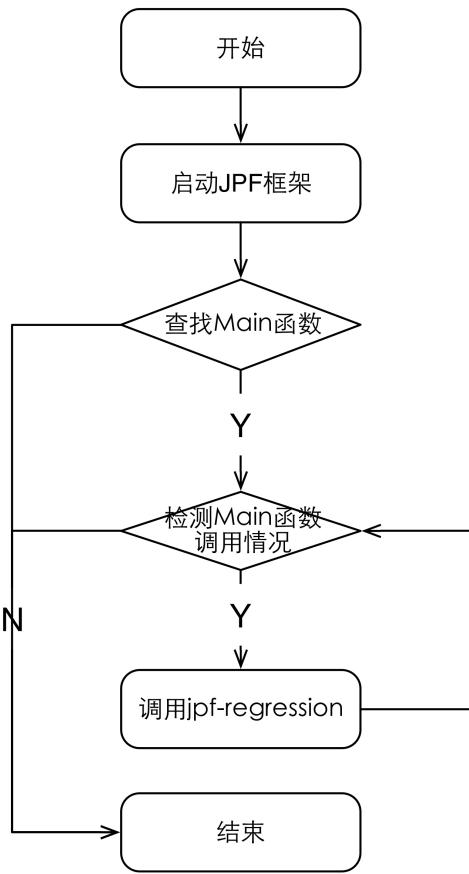


图 4.12 jpf 框架启动流程

在后续的冲突判定过程中，对于得到的冲突结果，我们需要对其追根溯源，挖掘其影响来源，以进行人工分析，判定该情况是否确实冲突。

因此，我们需要在变更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。

为了实现影响追踪系统，我们需要存储程序结构间的影响关系。如前所述，影响的来源主要有两类，即：

- 控制依赖
- 数据依赖

因而，为了描述该影响依赖关系，本文设计了 Dependency 类族，参见图4.14。该类中使用了一个二元组的数据结构 depend 来存储影响来源和受影响对象，并使用了多态机制来区分影响关系的类型，即是控制依赖还是数据依赖。Dependency 类族中重写了 hashCode() 方法和 equals() 方法，以便能够放入集合中进行存储。

影响追踪系统中具体使用到的数据结构可以参考表4.10。

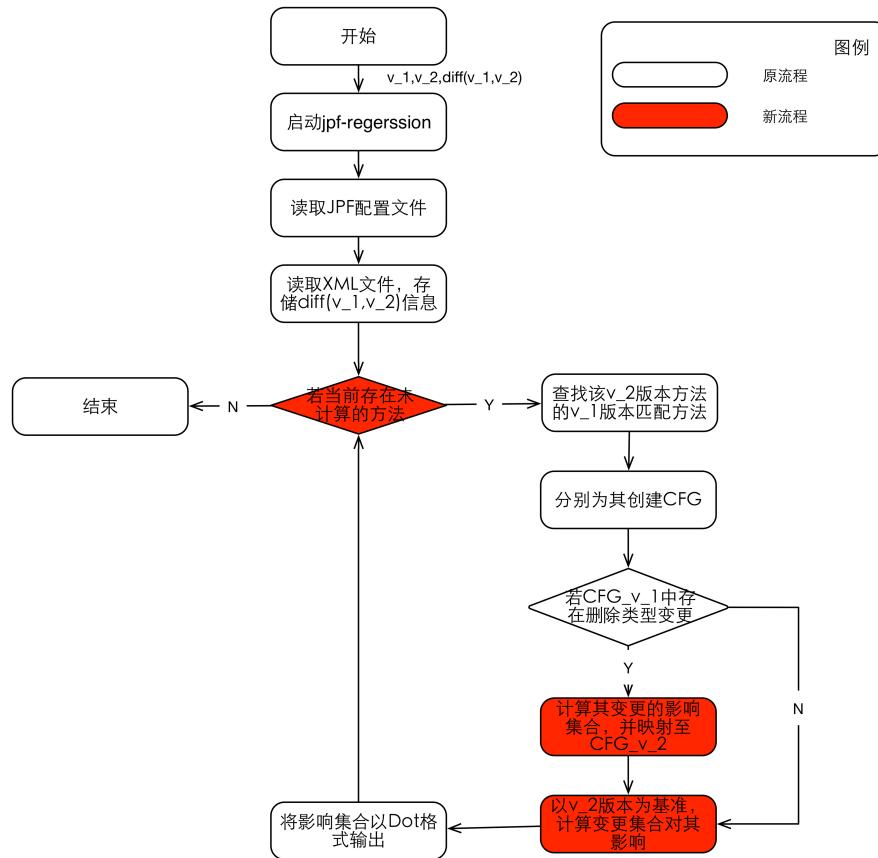


图 4.13 jpfr-regression 原流程及变化

表 4.10 影响关系数据结构

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

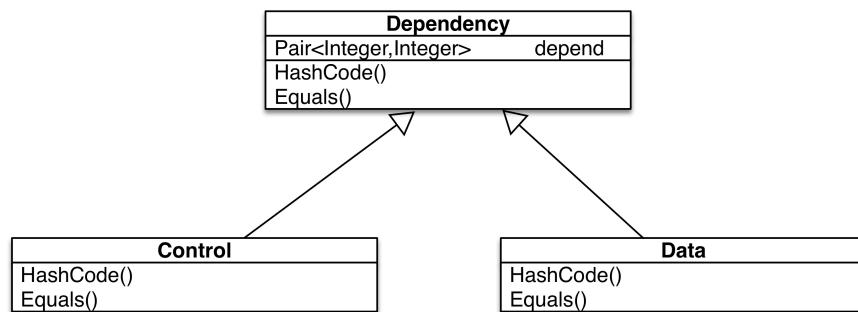


图 4.14 Dependency 类族

影响依赖关系的创建需要在进行变更影响分析过程的同时进行，以便记录下所有的依赖。最后将其输出到 Dot 文件中去，作为控制流承载的额外信息即可。

原有的 jpf-regression 工具由于是单次分析过程，因而一旦在运行过程中遇到问题，就会采用抛出异常终止运行的方式结束分析。然而我们在实际情况中需要进行大规模的分析作业，如果仅仅在其中单个文件的分析过程中出错就终止整个分析作业，会造成极大的时间和计算资源的浪费。

因而我们对该工具的异常处理方式进行了修改，使其在单次分析过程中如果遇到问题，则会及时抛出异常，但并不终止整个程序的运行，而采取了继续往下执行并分析其他文件的策略。然而分析错误是确实存在的，为了不丢失这类错误信息，我们为工具添加了错误记录系统，不断记录单次分析过程中遇到的问题。

我们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。该类的设计可以参考图4.15。其中使用到的数据结构可以参考表4.11的说明。

<b>ErrorCount</b>	
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 4.15 ErrorCount 类

原有的 jpf-regression 工具只能支持单次分析过程，在实际情况中我们需要工具具备大规模批量化分析的能力，以应对大规模软件系统的实际需求。为此我们可以保留原有的单次分析过程，然后在其上层进行封装，循环多次调用单次分析过程，以达到批量化自动分析的效果。这个过程由于进行了封装，对于用户而言是透明的。

同时，在进行大规模分析的时候，输出文件的命名格式也需要修改。在原单

表 4.11 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

次分析过程中，输出文件直接采用被分析的方法名进行命名。对于分析小型文件而言，这种设计就足够了，然而在大规模分析的时候，我们需要进行一定的优化。

由于大规模分析时，可能存在一些现象，例如：

1. 函数重载
2. 不同版本间的代码其方法可能无法一一匹配。例如有的方法仅在单个版本的代码中出现。

这些现象会使得工具中原有的输出文件命名方式不太合用。在这种情况下，我们采用的新命名格式为：

*MethodName + HashCode(MethodName) + ExtensionName*

其中，*MethodName* 即为方法名，无法保证方法名的唯一性。再利用 Java 中的 `HashCode` 方法，对 *MethodName* 计算其 `HashCode` 作为其后缀，以保证方法名的唯一性。最后 *ExtensionName* 即为文件扩展名，在 `jpf-regression` 中 *ExtensionName* = `.dot`。

同时我们也保留了原有的单次分析能力，以满足实际情况中的其他需要，例如进行小规模的案例分析。

其次，在进行大规模分析的情况下，由于实验数据量的庞大，我们无法按照单次分析过程中那样去人工查看并分析实验结果。因此，为了适应这种需要，我们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。

相关的类设计可以参考图4.16和图4.17。其中涉及的数据结构及其说明参考表4.12和表4.13。

表 4.12 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

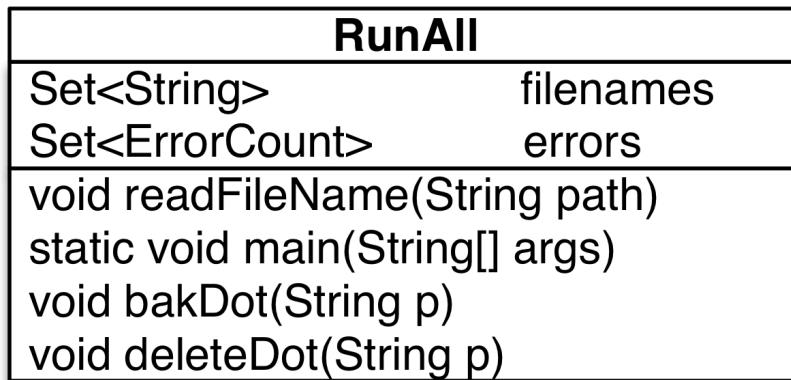


图 4.16 RunAll 类

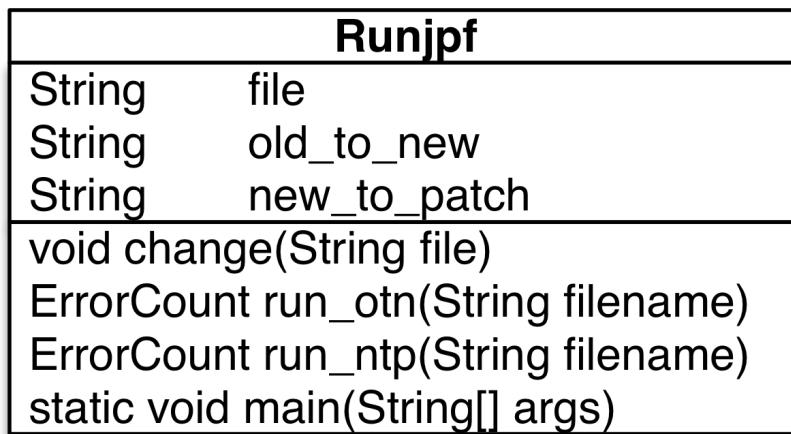


图 4.17 Runjpf 类

在实际使用 jpf-regression 进行实验的过程中，我们发现该工具存在一些 Bug，这些 Bug 或多或少的导致了分析结果的正确性和精度降低。我们对其中力所能及

表 4.13 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

表 4.14 Bug 报告

Bug	危害	修复
内部类无法进行方法匹配	小	否
只有只有单个版本代码中存在某个方法时无法进行方法匹配	小	是
将 $CFG_{v\_1}$ 的影响集合映射到 $CFG_{v\_2}$ 时判断条件出错	大	是
依赖 JAR 包 jpf_guided_test 出错	小	否
依赖 JAR 包 jpf_symboc 出错	小	否

的 Bug 进行了修复，并对这些 Bug 进行了总结。

目前已知的 Bug 及其修复情况可以参见表4.14。

## 4.4 冲突判定模块

冲突判定模块主要需要实现  $conflict$  函数的实际功能。

目前根据前文中所述的冲突分析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。

### 4.4.1 设计

在该模块的设计中，其输入输出过程可以描述如图4.18，输入输出的具体描述参见表4.15。

该模块的核心在于冲突分析算法3，因此，根据算法中的描述，该模块可以设计如图4.19所示。

可见，该模块的核心任务包括：

- 输入
- 输出
- 影响域计算：存储影响域信息，并计算重叠。

- 冲突分析：根据影响域重叠，对找到的冲突进行影响回溯。

因此，在冲突判定模块中，其流程可以设计如下：

1. 读取影响域分析模块的结果
2. 计算是否发生影响范围重叠
3. 对于发生了重叠现象的影响域，判定为冲突
4. 回溯冲突代码的影响来源并输出其影响依赖关系 *depend*
5. 根据得到的影响依赖关系进行人工分析，判定是否确实冲突

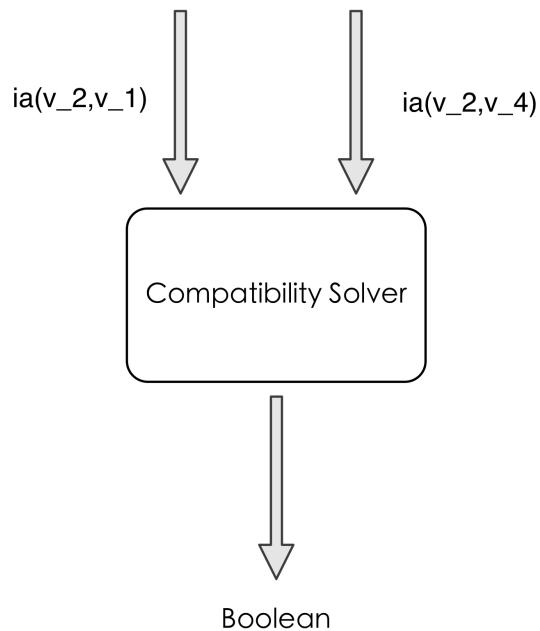


表 4.15 输入输出对照表

输入输出	描述
$ia(v_2, v_1)$	影响域分析模块的输出，语义影响域
$ia(v_2, v_4)$	影响域分析模块的输出，语义影响域
Boolean	是否发生语义冲突

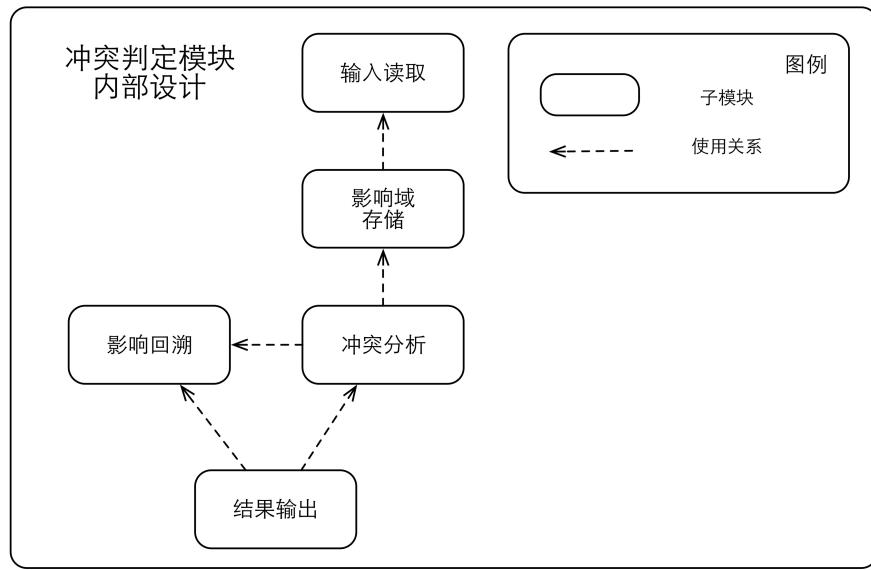


图 4.19 模块设计

#### 4.4.2 实现

该模块的实现主要参考了算法3的描述。在实际的实现中，其输入输出可以参考表4.16。

对于该模块所输出的语义冲突，需要进一步进行人工分析，判定是否确实存在冲突。人工分析的过程主要是参考输出的部分控制流中，被标注为冲突的节点是否确实发生了冲突。

表 4.16 输入输出对照表

输入输出	描述	格式
输入	$diff(v_2, v_1)$ 对于 $v_2$ 的语义影响域	Dot
输入	$diff(v_2, v_4)$ 对于 $v_2$ 的语义影响域	Dot
输出	语义冲突	Dot

可见，冲突判定模块中的主要工作包括：

- 计算影响域的重叠
- 对找到的语义冲突进行影响回溯，并以 Dot 格式将其涉及到的部分控制流和相关的冲突信息进行输出。

冲突判定模块中的类实现可以参考图4.20。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

并计算冲突的类。相关的数据结构说明参考表4.17

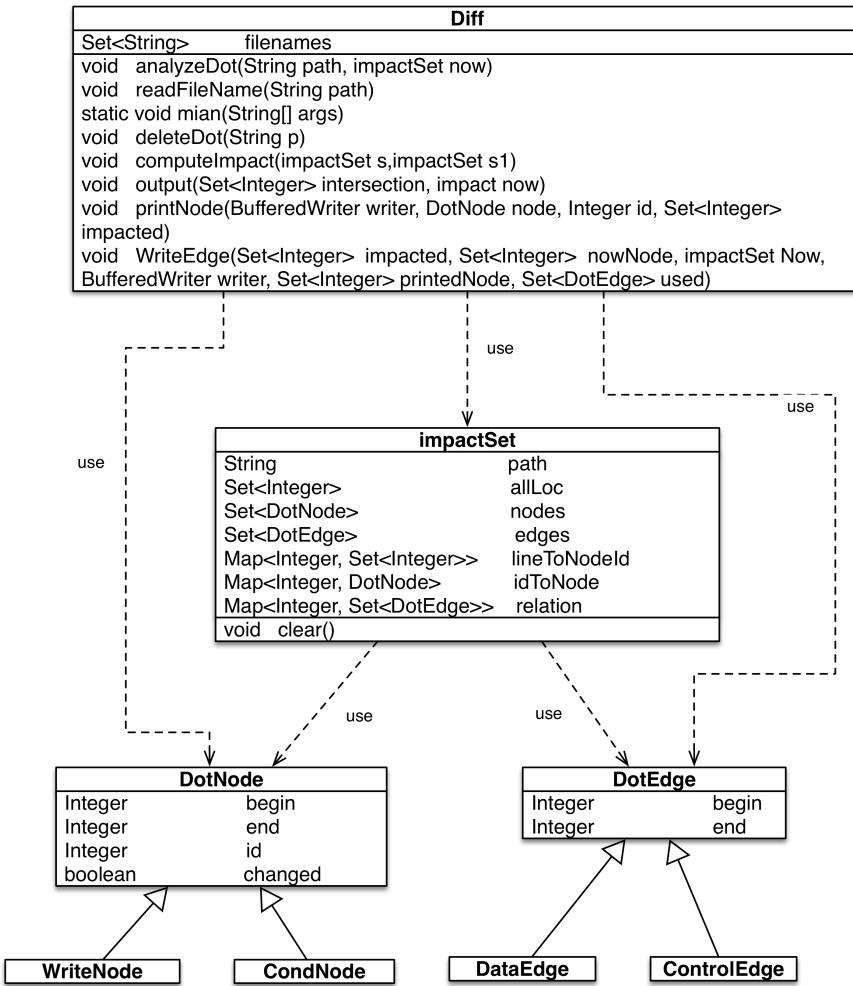


图 4.20 Diff 类族

表 4.17 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 4.18 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 4.19 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 4.20 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

该模块的实际工作流程可以参考图4.21。

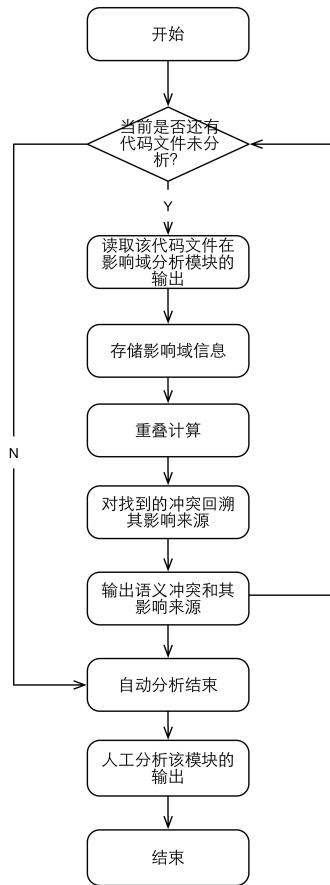


图 4.21 模块流程

## 4.5 本章小结

本章主要介绍了如何根据章节3.3中所提出的解决方案，完成补丁兼容性检测工具的设计与实现工作。

章节4.1中提出了工具的整体架构设计和运作流程。

章节4.2中介绍了版本迁移模块的设计与实现过程。

章节4.3中介绍了影响域分析模块的设计与实现过程。

章节4.4中介绍了冲突判定模块的设计与实现过程。

## 第5章 实验结果与分析

本章主要介绍如何对本文中所提出的补丁兼容性检测方法的工具实现进行实验，并且给出了实验结果和分析。

### 5.1 实验设计

考虑到本文中所要讨论的软件补丁兼容性检测问题，这是一个工业界中常见的实际问题，广泛存在于各类项目中。考虑到工业界中的实际情况，本文中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。如此一来，实验的结果就可以具有较强的说服力，能够回答本文中所提出的方法是否能够切实解决工业生产中面对的实际问题。

可见，实验应当回答如下的问题：

- 检测工具是否能够成功对实际项目进行分析？该项可以说明本文方法的可用性。
- 检测工具是否能够找到补丁的兼容性问题？找到的冲突是否存在误报的情况？该项可以说明本文方法的正确性。
- 检测工具能找到的兼容性问题有多少？该项可以说明本文方法的实用性。

因此，本章中应当对这些问题给予回答，并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5.1所示。



图 5.1 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

## 5.2 实验案例

为了使实验结果更具有说服力，本文在实验案例的选择中考虑采用工业界中的实际项目来进行实验，以测试检测工具的可靠性和可用性。

因此，本文中将采用 Eclipse JDT Core 项目作为实验案例。JDT Core 是 Eclipse 工具的 Java 基础组件，它主要提供了如下功能：

- Java 编译器。
- Java 文档模型。
- Java 模型。
- 编程帮助。
- 搜索。
- 源代码排版。

该项目的相关信息可以参见表5.1。

表 5.1 Eclipse JDT Core

信息	描述
语言	Java
文件数	约 1100 个
代码量	约 3W 行

Groovy-Eclipse 是一个 Eclipse 的插件集合，用于为 Groovy 项目提供 Eclipse 的工具支持。Groovy 是一门类似于 Java 的面向对象编程语言，Groovy 代码可以被编译器转化为 Java 字节码，从而在 JVM 上运行。由于这一特性，使得 Groovy 可以调用其他 Java 语言编写的库，从而大大丰富了其可用性。

Groovy-Eclipse 中提供了一个补丁后的 JDT Core 版本，该补丁主要用于增强 JDT Core 的功能，使其能够无缝编译 Groovy 代码。因而我们选择了该补丁作为实验数据中逻辑版本  $v_3$  的来源。

JDT Core 项目截止目前已推出发行版 4.4.2，我们从中选择了若干个发行版作为实验数据中逻辑版本  $v_1$  和  $v_2$  的来源。逻辑版本的具体说明可以参考表5.3，具体的实验数据选择可以参考表5.2，其中由于逻辑版本  $v_4$  由版本合并而来，因此在表5.2中不予列出。

从表5.2中可见，我们一共选择了 7 个发行版本作为逻辑版本  $v_2$  的来源。也就是说，我们将以固定的逻辑版本  $v_1$  和  $v_3$  为基准，并选择不断变化的逻辑版本  $v_2$  来进行实验。

表 5.2 实验数据

代码	发行版本	逻辑版本
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 5.3 逻辑版本对照表

逻辑版本	描述
$v_1$	旧版本
$v_2$	新版本
$v_3$	应用补丁 $p_1$ 后旧版本
$v_4$	版本 $v_2$ 和版本 $v_3$ 合并后版本，相当于应用补丁 $p_1$ 后新版本

这样做好处是我们可以更加清晰的了解本文中所提出的补丁兼容性分析方法，包括：

- 是否可针对工业界实际项目进行分析？
- 是否切实贴合工业界的实际需求？
- 是否对同一软件系统的多个不同版本具有普遍适用性？

### 5.3 实验结果与分析

由于检测工具由多个模块构成，因此实验是分步进行的，本节中给出每个模块的输入输出数据，并对其结果加以分析。

#### 5.3.1 版本迁移模块

根据我们所选择的实验案例，其所有版本的提交与合并过程可以参考图5.2。

可见，该过程中我们以 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 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，用于各自实现版本合并过程。

表 5.4 版本合并结果结果

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

在版本合并的过程中，我们检测到的待解决冲突数量可以参见表5.4。通过 Beyond Compare 工具，这些冲突都能够得到解决。可以发现，冲突最少的是版本 4.3.x，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，冲突数量陡增，这可能是由于版本升级较大的缘故。

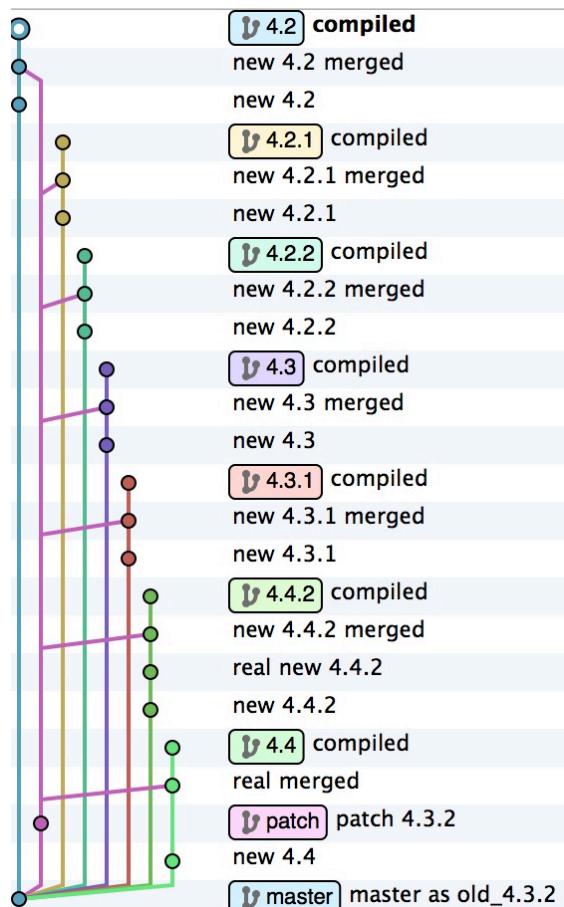


图 5.2 git 版本合并

在实际操作过程中，由于后续的影响域分析模块需要提供 Java 代码编译后产

表 5.5 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

生的 Class 文件，我们对于合并后的版本  $v_4$  还需要进行编译。实验过程中，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5.5。对比编译过程中的错误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。

如上所述，我们的版本迁移模块是可用且有效的。

### 5.3.2 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对其结果进行分析。

#### 5.3.2.1 差异性分析模块

在差异性分析模块中，我们比较关注预处理算法的结果以及能够成功完成差异性分析的文件数量。

对于预处理算法而言，实验结果如表5.6所示。

表 5.6 Program Differ 结果

代码	发行版本	预处理数	$diff(v_2, v_1)$ 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

可见，我们的预处理算法对原有的 ASTro 工具的直接输出结果进行了有效的过滤。预处理算法的好处在之后的影响分析模块中体现得更为明显。

整个差异性分析模块的输出结果如表5.7和表5.8所述。

表 5.7 Program Differ 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 5.8 Program Differ 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

可见，绝大多数的文件都能够成功的进行差异性分析。

### 5.3.2.2 影响分析模块

在影响分析模块中，我们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5.9所述。

表 5.9 impact analyzer 结果

代码	$v_2$	分析结果数
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

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5.10所述。

表 5.10 impact analyzer 结果

代码	$v_2$	分析结果数
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

### 5.3.3 冲突判定模块

在冲突判定模块中, 应用本文提出的冲突分析算法后, 再通过影响追踪系统进行辅助人工分析, 可以得到如表5.11的结果。

表 5.11 分析结果

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，在不使用预处理器的情况下，实验结果如表5.12所示。

表 5.12 分析结果

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

如表所示，误报的影响域重叠数量陡增，这主要是由于差异性分析模块的误差所导致的。可见，选择正确的差异性分析算法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的体现。

同时可以发现，由于运用了预处理算法，这里找到的冲突结果都是由于对不同行的变更影响到了相同的代码行而被挖掘出来的。事实上，如果存在对同一代码行的变更，其所导致的变更在版本合并的过程中就会暴露出来。

从以上结果中可以发现，对本文中所提出的兼容性检测工具而言：

- 能够成功的将为某个专门版本代码而设计的补丁应用于其他版本代码上。
- 确实能够成功的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。
- 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文件的代码中，很难直接被肉眼所发现。
- 目前能够找到的语义冲突数量较少，这可能是由于：
  - 确实只有少量语义冲突存在。
  - 可能受限于工具的精度不够高、分析的变更影响范围不够广等因素。有待以后的工作中做进一步的实验。

因此，本文中所实现的兼容性检测工具对于工业界的实际问题来说是可用的、正确的，然而其实用性还有待进一步的提高。

## 5.4 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5.1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5.2中主要介绍了实验案例的选取过程。

章节5.3中主要介绍了实验的结果和对结果的分析。

## 第 6 章 结论

### 6.1 工作总结

首先，本文提出了软件补丁兼容性检测的问题，并给出了详细的形式化定义。

其次，为了解决这样一个问题，本文通过对该问题进行深入的分析和讨论，提出了一套通用的补丁兼容性分析解决方案，它包括以下部分：

- 补丁应用
- 语义影响域分析
  - 程序间差异性分析
  - 变更影响分析
- 冲突分析

其中补丁应用过程中的 *merge* 函数和语义影响域分析的两个子过程 *diff* 函数和 *impact* 函数可以自由使用符合要求的相应算法实现，以提高解决方案的实用性。

目前冲突分析过程提出了一种较简单的自动冲突分析算法作为 *conflict* 函数的实现，更精确的分析结果目前需要人工分析的辅助，为此解决方案中需要变更影响分析过程提供影响追踪函数 *impact\_track* 来追溯语义影响的来源。

最后，本文对该套解决方案给出了具体的工具设计和实现方案，并对该套兼容性检测工具在 Eclipse JDT Core 项目上进行了测试，发现该工具是确实可用而有效的。它确实能够挖掘出补丁间的语义冲突并向用户进行报告。

可见，本文的主要成果包括：

- 提出了软件补丁兼容性检测的问题。
- 提出了一套补丁兼容性分析的通用解决方案。
- 根据提出的解决方案，实现了具体的兼容性检测工具，并在中型项目 Eclipse JDT Core 的八个不同版本上进行了实验，论证了该解决方案对工业界实际项目的可用性、正确性和实用性。

### 6.2 未来工作

对于本文中所提到的补丁兼容性解决方案和其工具实现，其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工具进行进一步的实验，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] 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

感谢贺飞老师对我的悉心指导，他不仅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，其言传身教更是令我受益匪浅。

其次要感谢实验室的同学和师兄等的帮助，如郭心睿、刘盛鹏、周旻等，他们对我提供了很多帮助，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。