

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

Software Patch Compatibility Checking Based On Impacted Area 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

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。

为将软件应用于某些特定场景，常常需要开发专门的补丁。伴随着软件本身的不断演进，为每个软件版本开发专门的补丁耗费的人力物力极大。本文研究在软件演化的背景下，不同版本软件的补丁能否共享。

在解决该兼容性问题的过程中，本文的主要贡献在于以下几点。

首先，本文对软件补丁兼容性问题进行了分析和讨论，发现该问题的难点在于如何找到变更之间的语义冲突。

其次，为了解决该问题，本文提出了一套基于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。该检测方案主要包括两个步骤。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，完成补丁间的语义冲突检测。

最后，通过在 Eclipse JDT Core 项目上的实验，我们发现该工具确实能够发现软件补丁的兼容性问题，说明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；变更影响域；补丁；兼容性检测；软件演进

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 is always evolving. In practical application, patch is one of the most common program which is used for software evolving, like enhancing its functionality, fixing bug and so on.

For the purpose of applying software under specific circumstances, a special patch is needed. But during the whole evolving process of the software, developing this special patch for each version of the software is too costly. Under the circumstance of software evolving, This paper focus on the problem of sharing the patch between different software versions.

The major work and contribution of this paper are as follows.

First of all, after analyzing the problem, we found out that the core of this problem is how to find out the semantic conflicts between the changes.

Secondly, we have proposed a solution called software patch compatibility checking based on impacted area analysis for this problem and developed a corresponding tool. The solution 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 patches.

And at last, 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

第 1 章 绪论	1
1.1 研究背景	1
1.2 研究内容	2
1.3 本文组织结构.....	3
第 2 章 相关工作	4
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
第 3 章 软件补丁兼容性检测方法	10
3.1 兼容性问题.....	10
3.2 检测方法概述.....	10
3.3 本章小结	12
第 4 章 软件变更影响域分析	13
4.1 程序间语法差异性分析.....	14
4.1.1 相关定义.....	14
4.1.2 分析方法.....	14
4.1.3 模块设计与实现	15
4.2 变更语义影响分析	19
4.2.1 相关定义.....	19
4.2.2 分析方法.....	20
4.2.3 模块设计与实现	21
4.3 本章小结	28

目 录

第 5 章 软件变更冲突检测方法	29
5.1 相关定义	29
5.2 分析方法	29
5.3 模块设计与实现	30
5.4 本章小结	33
第 6 章 实验结果与分析	34
6.1 实验设计	34
6.2 实验案例	35
6.3 实验结果与分析	36
6.3.1 版本迁移	36
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	44
6.4 本章小结	46
第 7 章 结论	47
7.1 工作总结	47
7.2 未来工作	47
参考文献	49
致 谢	51
声 明	52
个人简历、在学期间发表的学术论文与研究成果	53

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^[1]。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发生更新行为。这是软件维护周期中的常见活动^[2]。为了进行软件版本演进，现代软件工程提出了许多解决方案。

补丁（Patch）就是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方案。补丁在工业界中得到了广泛的实际应用，成为软件维护过程的重要组成部分^[3]。

补丁一般通过 Diff 工具产生文本数据^[4]，用于表示以行为单位的程序间的语法差异。虽然得到了广泛应用，但补丁程序仍然具有一定的局限性。

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。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。然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，对时间和人力成本造成了许多浪费。

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。

由该场景可见，软件补丁的兼容性检测是一个具有实际意义的问题。学术界目前尚无这方面的相关工作。

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）^[5]，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。

事实上，由于软件代码的耦合性，单行变更就足以影响到软件系统的其他部分^[6,7]。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。

考虑到该问题主要是由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，该问题的解决需要找到这些冲突。这种冲突既可能是语法结构上的，也可能是语义层面上的。因此，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，该问题主要关注的是在应用过程中是否会造成功能结构上的错误。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。如果能够将补丁 p 成功应用于新版本，则认为该补丁是可以在语法上兼容于新版本的。对此，现有的版本控制系统等工具就足以完成。

从语义角度出发，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，就语义层面而言，需要保证补丁 p 在新版本中引入的语义不会影响到从旧版本演进到新版本时其变更所引入的语义变化，即需要保证这些变更之间不存在语义上的冲突。目前，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。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软件的语义冲突，本文提出了一种基于语义影响域分析的冲突检测方法，该方法可以分析软件代码在不同版本间所引入的变更，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。通过分析这些语义影响域之间是否存在重叠，能够发现补丁对旧版本造成的语义变化是否会影响到从旧版本演进到新版本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案，该方案能够分析软件版本之间的变更，并找到变更影响域，通过对变更影响域之间是否重叠进行判断，就能够找到可能存在语义冲突的代码位置。该套解决方案主要包括：

- 软件变更影响域分析：分析并获取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：根据得到的变更影响域，分析其是否存在语义冲突。

最后，本文根据该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。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，主要包括构成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^[8]。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^[9,10]。

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^[11,12]。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的影响的技术集合^[13]，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^[14]。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。

变更影响分析方法的分类：

- 基于追踪性的变更影响分析^[15]，它会追踪两个不同抽象级别上的元素间的依赖性，其目的在于链接不同类型的软件工件（需求、设计等）。
- 基于依赖关系的变更影响分析^[16]，它致力于衡量变更的潜在影响，并尝试去分析程序的语法关系。这些语法关系表示程序实体间的语义依赖关系。这类变更影响分析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，变更影响分析的目标就在于找到这些副作用（Side Effect 或者 Ripple Effect）^[17]，并防止之。变更影响分析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^[15,18]，更详细的流程可以参考图2.1。

1. 确定变更集合，此时需要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^[19]。
2. 衡量变更集合引入的影响。这一步是目前大部分变更影响分析技术的重点。

若按分析技术的类型进行划分，则主要的两类包括静态分析和动态分析：

- 静态分析：包括历史分析、文本分析、结构分析等^[20,21]。静态分析通过分析程序的语法、语义或者历史依赖来实现，通常会产生很多误报（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
- 动态分析：在线分析和离线分析。它考虑特定的输入，并且依赖程序运

行时收集到的信息来完成分析（如运行时的追踪信息、覆盖信息等）^[22]。其得到的影响集合往往比静态分析的结果具有更高的精度，但其开销也相应更大，而且经常包括错报（False Negative）。

- 在线变更影响分析使用程序运行时收集的信息来进行

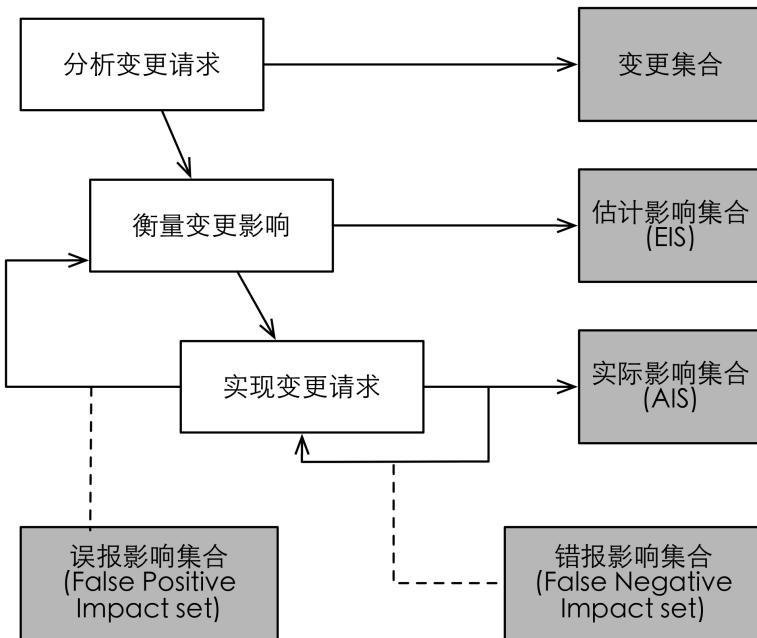


图 2.1 变更影响分析过程

近年来，涌现出一些以变更影响分析技术作为支撑而实现的工具，它们一般在实现变更影响分析技术之后，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^[23]。

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^[24,25]。

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^[26]。

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^[27]。

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。

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。变更影响分析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变更影响分析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, 以及软件系统的大和复杂性越来越高的因素, 人们对于有效的变更影响分析的需求越来越高。^[28]

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会通过分析程序组件间的内部关

系来衡量变更的影响。这类技术能够用程序的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，并且能够将需要关注的范围缩小，使得只需关注受到影响的程序行为集合即可。

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

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部分的影响，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，并生成相应的路径条件 (Path Condition)，这些路径条件可以描述受到影响的程序行为。在分析完毕之后可以利用 SMT 等技术将其求解，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，它的变更影响分析过程的主要特点在于：

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并计算的范围，最后得到变更对其所处的方法内部的其他语句所造成的影响。也就是说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进行迭代。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。

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^[30]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 软件补丁兼容性检测方法

3.1 兼容性问题

回顾1.1中提到的应用场景：

- 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，在集成该新版本的时候，想知道原补丁是否还能继续使用。

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。在集成该新版本的时候，原补丁是否能够继续使用的问题就可以转化为原补丁是否会和升级补丁产生冲突的问题。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，补丁兼容性问题就得到了简化。我们可以找到每个补丁中的变更所影响到的程序语法结构的集合，并探讨这两个集合之间是否存在一定的交集，显然，如果两个补丁中的变更影响到了相同的语法结构，该位置上的语法结构受到了双重影响，该位置可能出现冲突。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。

3.2 检测方法概述

本文中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，我们需要找到补丁中的变更所影响到的其他语法结构，也就是后文中所提出的变更影响域的概念。最后，我们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。

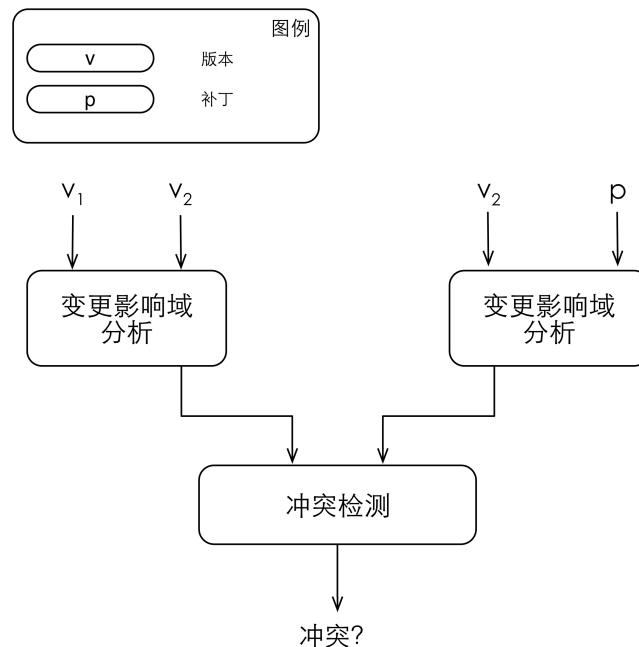


图 3.1 解决方案

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 应用至代码版本 v_2 ，以便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 ，获得版本 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. 解决分支合并中可能出现的冲突问题。

最后所得到的 v_4 版本代码即为我们所需要的在版本 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版本的代码间的变更集合，并以此为依据进行分析，最后找到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合影响的程序语法结构。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。

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：

- 影响域分析模块：实现解决方案中的软件变更影响域分析过程。
- 冲突判定模块：实现解决方案中的软件变更冲突检测方法。

这些模块的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及其检测方法。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

本章中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。

如前所述，我们需要对补丁中的变更进行变更影响域分析，以此来找到变更对应的语义影响域，即所谓的变更影响域，用于进行后续的冲突检测。

软件变更是对程序代码结构所作出的修改，它会将原有的代码结构变更为新的代码结构。因此，由于修改前的代码结构与软件中其他代码结构的耦合性，在应用软件变更之后，会导致这些相关的代码结构的行为受到该软件变更的影响，具体是什么影响则需要根据变更的语义来判断。

我们将这些受到补丁中变更影响的代码结构集合称之为变更影响域。其更详细的定义可以参考后续的相关定义。

为了找到所谓的变更影响域，我们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语法差异性。用于寻找变更的影响域。
2. 变更的影响域，即受变更集合的语义影响的其他程序语法结构。用于确定变更造成的语义影响。

在有了这两类集合之后，我们就知道了软件变更对代码的语义影响域，我们可以将这整个过程称为变更影响域分析。

可见，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，用于判断变更影响域之间是否会产生冲突。

因此，该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，变更影响域分析会根据找到的软件变更集合，分析并得到变更影响域。

如上所述，那么软件变更影响域的过程也就可以相应地拆分为两个子分析过程。首先，使用程序间语法差异性分析来完成寻找软件变更集合的过程，其次，使用变更语义影响分析来完成寻找变更影响域的过程。

因此，该过程所对应的影响域分析模块在实现的过程中也将对应地拆分为两个子模块：

1. 差异性分析模块。该模块将实现程序间语法差异性分析的过程。
2. 影响分析模块。该模块将实现变更语义影响分析的过程。

由于这两个子分析过程已经有许多相关的工作出现，因此我们在实现检测工具中对应的影响域分析模块的时候将采用已有的成熟算法来完成。工具实现的主要精力将放在子模块的整合和调整过程上。

下面将分别对这两个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用于完成寻找不同版本间的软件变更的过程，该过程中需要分析代码间的语法结构的差异性，并得出相应的变更集合。可见，在该过程中，我们主要关注程序间的语法结构差异性。

近年来程序间语法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其所对应的差异性分析子模块。

4.1.1 相关定义

本节中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的定义，以便能够更清楚的了解该分析过程的实质。

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況而变化，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： $\mathcal{P} : S \times S$ 。 \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上的差异信息。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 `.patch` 类

型的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，该分析过程应该满足如下需要：

- 输入为两个不同版本的源代码。
- 输出为源代码间的软件变更集合。
- 每条变更描述对于该语言的合法语法结构的变更。

除此以外，该分析过程还应当描述的信息包括：

- 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等。让我们能够知道该变更的作用位置等。
- 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，让我们能够知道该变更的作用范围。

选择这样的分析结果类型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 *.patch* 类型的补丁文件只描述单纯的文本行的变更，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，因此该类型的差异性分析方法是无法满足我们的需求的。

在实践过程中，一种比较好的选择是在抽象语法树 AST (Abstract Syntax Tree) 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的相关信息。

4.1.3 模块设计与实现

本文中差异性分析模块在实现中采用了 *jpf-regression* 工具自带的前置工具 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 代码的比对。它会比对两个文件的抽象语法树，并从中抽出对应的不同之处，形成语法结构上的差异性，并将变更前后的 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格式进行输出，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，我们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，受限于 ASTro 工具的具体实现，其输出结果存在着一定的问题，因此我们对 ASTro 工具的输出结果进行了一定的过滤。其输出结果的问题主要包括：

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中的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，我们无法解决，不过在实践中这只是极少数现象。

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。

该过滤算法可以用伪代码1进行描述。

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

我们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，我们的预处理操作是正确的，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，我们完成了分析过程的自动化。

在实现该模块的时候，我们采用了 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。
- 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4.1。

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 4.2 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

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

而且，我们还采用了 shell 脚本完成对后续分析过程的支持，使其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，通过类似键值对的方式定义了各项属性的值，可以参考表4.2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。因而在进行变更语义影响分析时，我们需要进行相应配置，使得：

- 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”。

在实际操作中，我们只需做这样的设置即可。

最后，整个差异性分析模块的工作流程可以参考图5.1。



图 4.1 程序间差异性分析流程

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。在该过程中，我们主要关注的是语法结构之间在语义上的相互影响。

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语义影响分析算法作其对应工具模块的具体实现。

4.2.1 相关定义

本节中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。

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。

事实上，影响是由于代码之间存在着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。可见，这里所谓的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。

有了影响的定义以后，我们就能得到变更影响域的概念。前文中所提到的变更影响域，也就是变更的语义影响域（Sematic Impacted Area）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。影响范围实际上描述了变更所造成影响的传播范围。

因此，变更影响域可以较形式化的定义如下：

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， $p \subset \mathcal{P}, v \subset C, s_k \subset \mathcal{S}, k \in \mathbb{N}$ 。这里的 s_k 其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^[31]。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。

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。

因此，如前文所述， $impact$ 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，也就是我们所需要的变更影响域。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。

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域。
- 计算过程中可以指定受影响的范围和语法结构类型。用于设置分析的精度。
- 具有影响追踪系统，将计算影响域的过程进行记录。方便后续的冲突检测过程回溯影响的来源。

4.2.3 模块设计与实现

在实现该模块的过程中，本文主要采用了 jpf-regression 工具的变更语义影响分析算法。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语义影响分析。在实现过程中，我们采用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。

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 (Control Flow Graph)，而变更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。可见，该模块实际上将影响域信息输出为图形表示。

因此，该模块中输出的影响域的相关信息主要包括两类：

- 受影响的 CFG 节点。这实际上就是影响域中的元素。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来源，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。因而在实践过程中，我们可以重用 jpf-regression 的代码，并对其进行改造，使其符合本文的实际需要。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符合，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4.3进行对比。首先我们去掉了图4.2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4.3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并计算过程中增加了影响追踪系统。

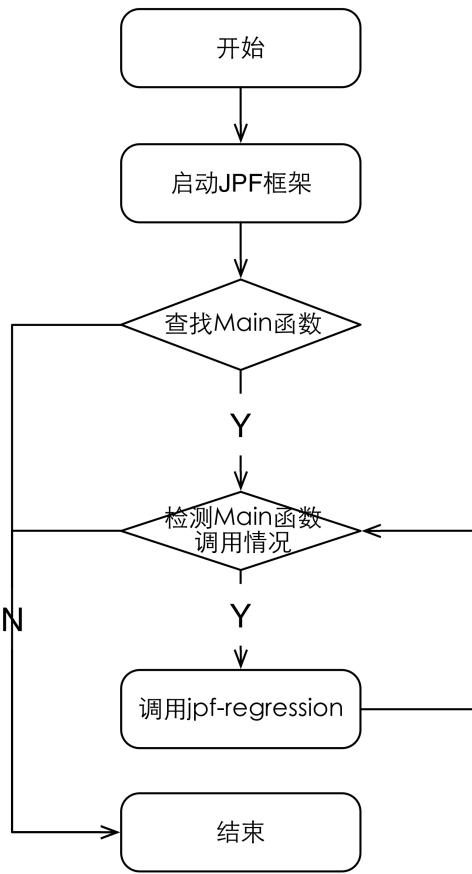


图 4.2 jpf 框架启动流程

在后续的冲突判定过程中，对于找到的可能发生冲突的位置，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。

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，为了描述该影响关系，本文设计了 Dependency 类族，参见图4.4。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4.3。

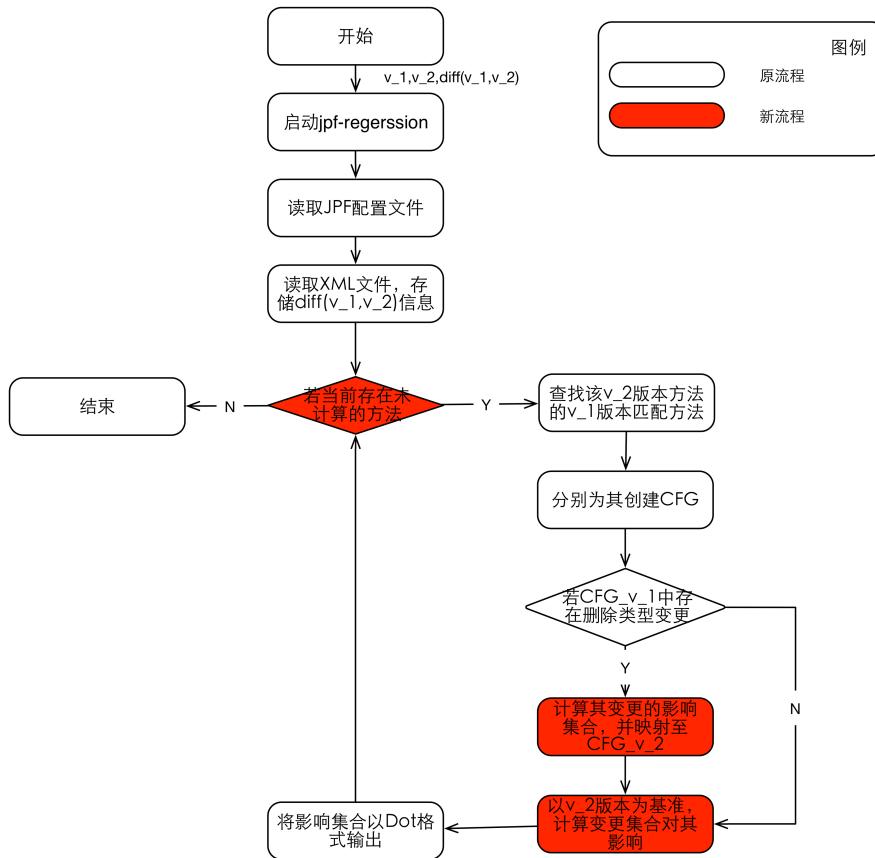


图 4.3 jpf-regression 原流程及变化

表 4.3 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

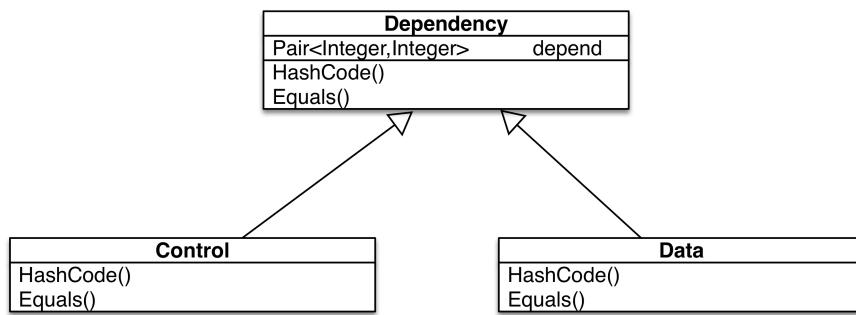


图 4.4 Dependency 类族

影响关系的创建需要在进行变更语义影响分析过程的同时进行，以便记录下所有的依赖。最后将其作为影响域相关信息的一部分进行输出。在实现过程中，将其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4.5。其中使用到的数据结构可以参考表4.4的说明。

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

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。

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

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* 由即为方法名，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。

最后，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。

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地址，然后循环调用 `Runjpf` 对象进行单次分析过程。

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

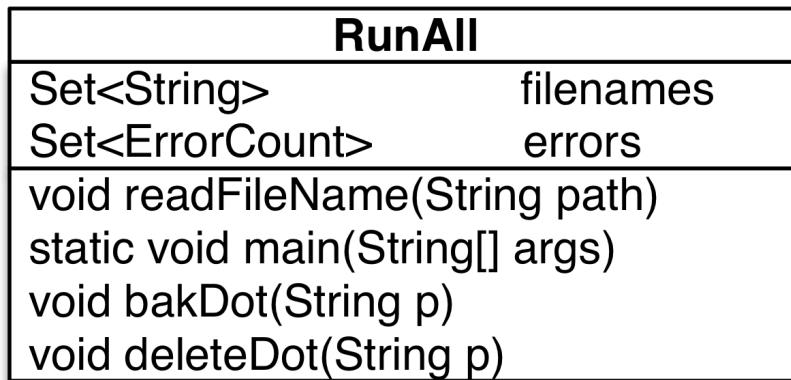


图 4.6 RunAll 类

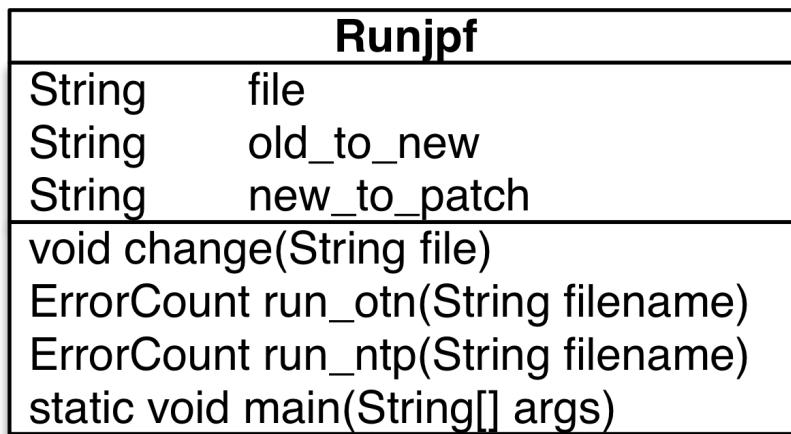


图 4.7 Runjpf 类

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 4.6 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

表 4.7 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4.7。

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。通过判断变更影响域之间是否存在重叠，本文中实现了简单的自动分析算法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。

下面将进行具体的介绍。

5.1 相关定义

为了进行变更的冲突检测，我们需要知道什么是冲突。以下所指的冲突都指语义冲突。

根据前文中的讨论，我们可以发现，冲突是实际上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影响域发生了重叠。因此，冲突可以形式化的定义如下：

定义 5.1： $conflict : \mathcal{P} \times \mathcal{P} \mapsto T, F$ 。 $\forall p_i, p_j \in \mathcal{P}$ ，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 $v \in C, i, j \in \mathbb{N}$ 。即如果两个变更集合对应的变更影响域产生了交集，那么就发生了冲突。

可以看出，冲突也就是由于两个变更集合中的不同变更，直接或间接的影响到了相同的语法结构而造成的。由此可见，*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。我们将对比两个版本代码的变更影响域，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可能会发生冲突的。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，而产生误报的情况。

显然，如果重叠不存在，则补丁间的兼容性是可以得到满足的。而对于如何界定重叠部分代码是否会引发补丁间的冲突而言，人工分析的介入是必须的，因

为这部分代码的冲突判定与补丁的变更目的密切相关。而我们无法从代码中直接获取到这种信息，因此只有依靠外界来提供并以此为依据判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，而且还需要知道是哪些变更影响到了这部分代码，因而我们需要影响追踪系统提供回溯功能。由于影响追踪系统会记录变更语义影响分析过程的轨迹并存储了程序语法结构间的影响关系，因此，我们只需要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。可见，人工分析的过程中主要需要影响追踪系统中的回溯模块提供支持。

该分析过程可以参考算法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  $\leftarrow$  True
3: else
4:    $s \leftarrow s_1 \cap s_2$ 
5:   if  $s = \emptyset$  then
6:     result  $\leftarrow$  True
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。

该模块的核心任务包括：

- 输入：读取软件变更影响域分析过程的输出，即变更影响域
- 输出：找到可能发生冲突的代码位置，并输出其影响来源。
- 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。

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. 计算是否发生影响域重叠
3. 对于发生了重叠现象的影响域，判定为可能发生冲突
4. 回溯冲突代码的影响来源并输出其影响关系
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

该模块的实现主要参考了算法2的描述。由于其输入为影响域模块的输出，因此它在输出影响关系的时候，可以给出和影响域模块相类的结果，即将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冲突的类。相关的数据结构说明参考表5.1

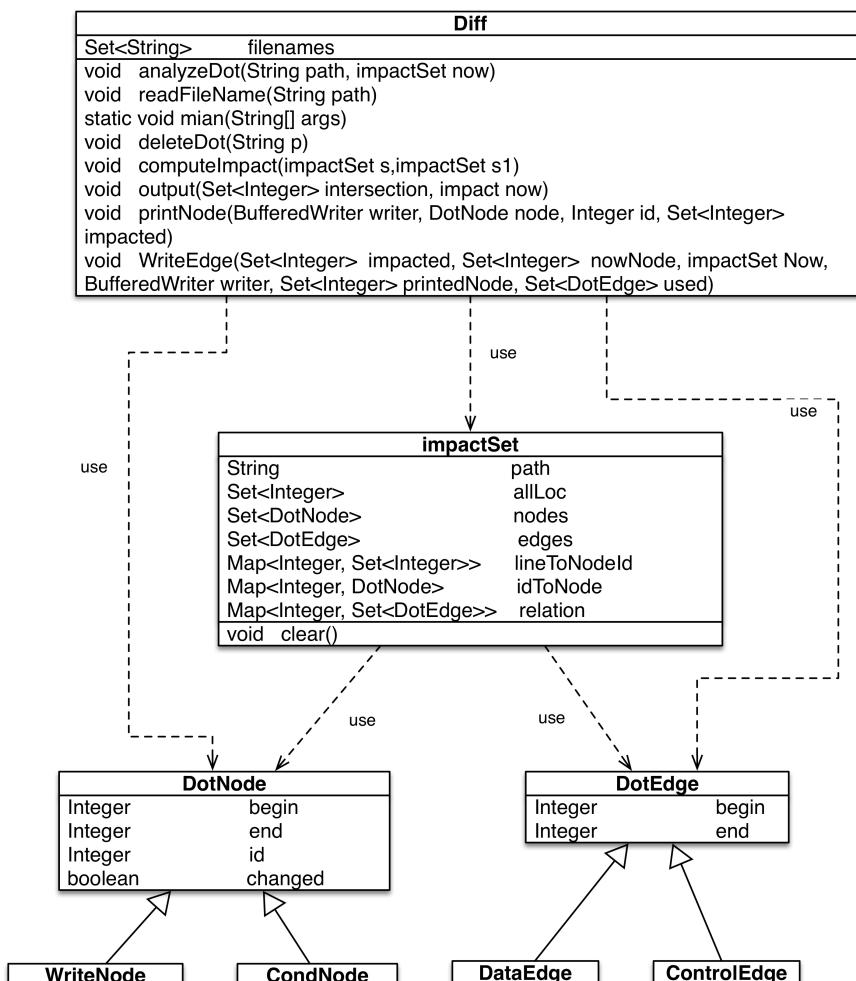


图 5.1 Diff 类族

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

该模块的实际工作流程可以参考图 5.2。

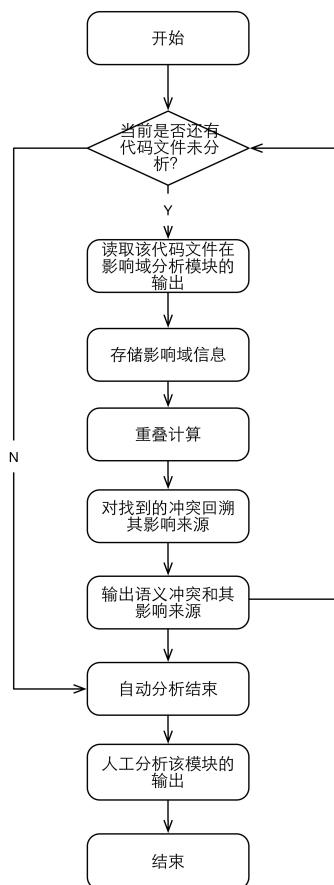


图 5.2 模块流程

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本文所提出的补丁兼容性检测工具进行实验，并且给出了实验结果和分析。

6.1 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，能够说明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，实验应当具备如下的目的：

- 判断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。
- 判断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。
- 判断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，本章中应当进行满足要求的实验，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

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。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 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6.3，具体的实验数据选择可以参考表6.2，其中由于逻辑版本 v_4 由版本合并而来，因此在表6.2中不予以列出。

从表6.2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。

这样做的好处是我们可以对本文中所提出的补丁兼容性检测工具进行详尽的测试，包括：

- 测试该工具能否针对工业界实际项目进行分析。
- 测试该工具能否切实贴合工业界的实际需求。
- 测试该工具能否对同一软件系统的多个不同版本具有普遍适用性。

表 6.2 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 6.3 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

6.3 实验结果与分析

由于检测过程由多个步骤实现，因此实验是分步进行的，本节中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，本文中首先需要完成将补丁 p 应用于版本 v_2 的过程。该过程中我们采用了 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。

根据我们所选择的实验案例，所有版本的迁移过程可以参考图6.3。

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。

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6.4。通过 Beyond Compare 工具，这些冲突都能够得到解决。

根据图6.2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

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

分比甚至高达 20% 至 30%，这可能是由于版本升级较大的缘故。

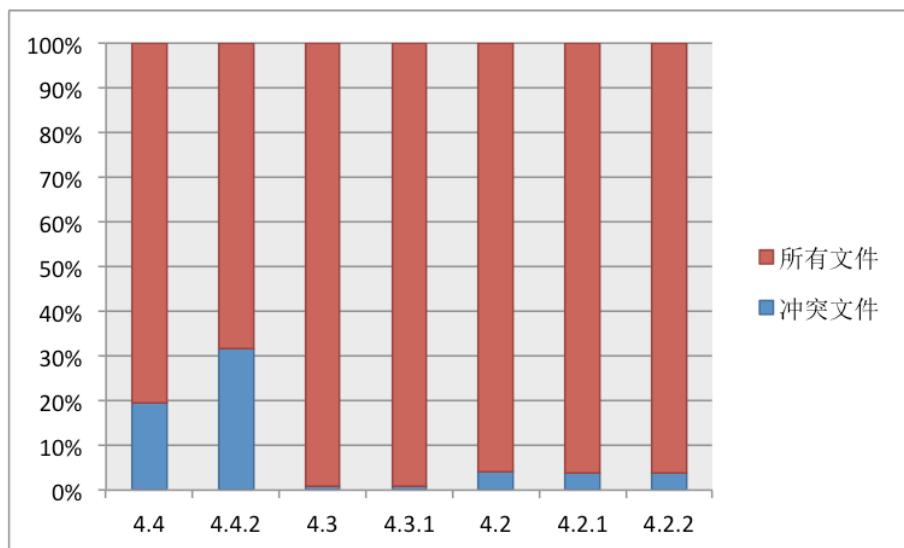


图 6.2 版本合并结果

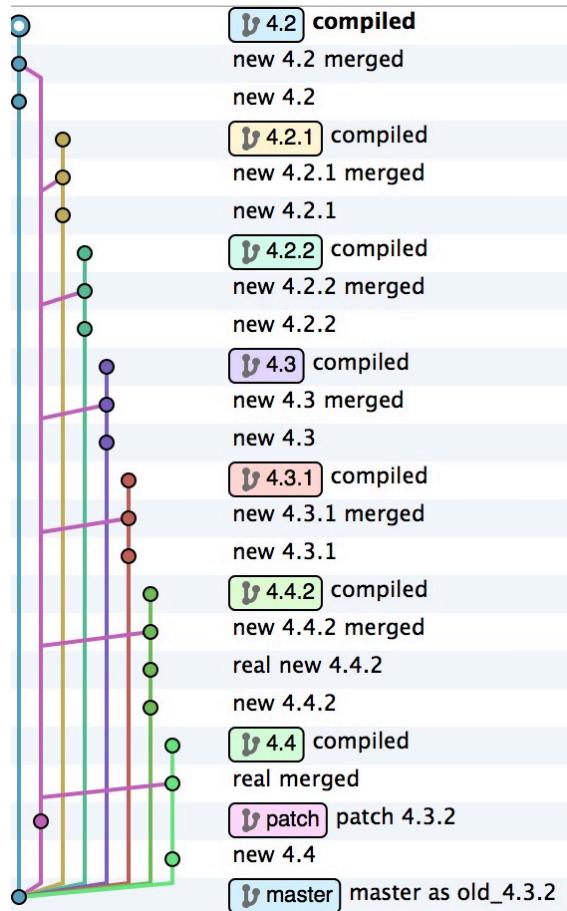


图 6.3 git 版本合并

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6.5。

同样，从图6.4可以看出，编译失败的文件，其所占的百分比很低。

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

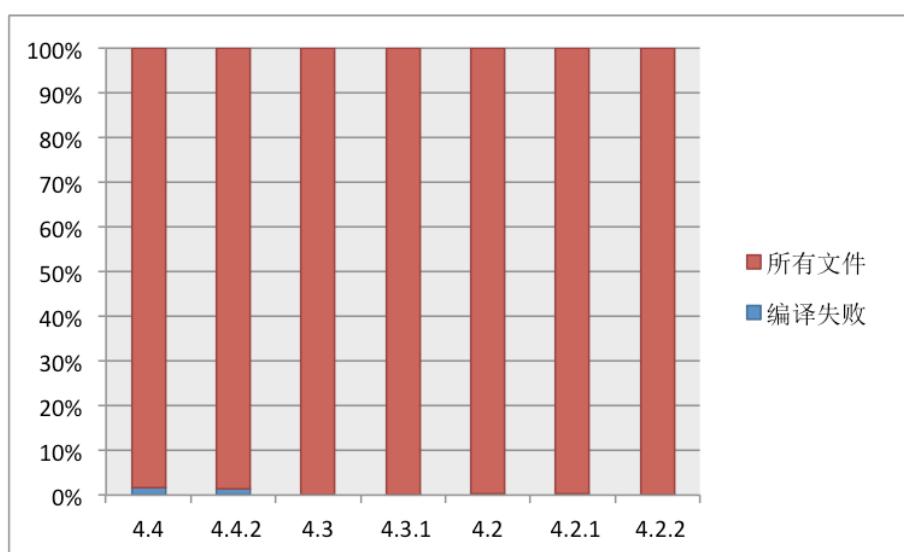


图 6.4 编译结果

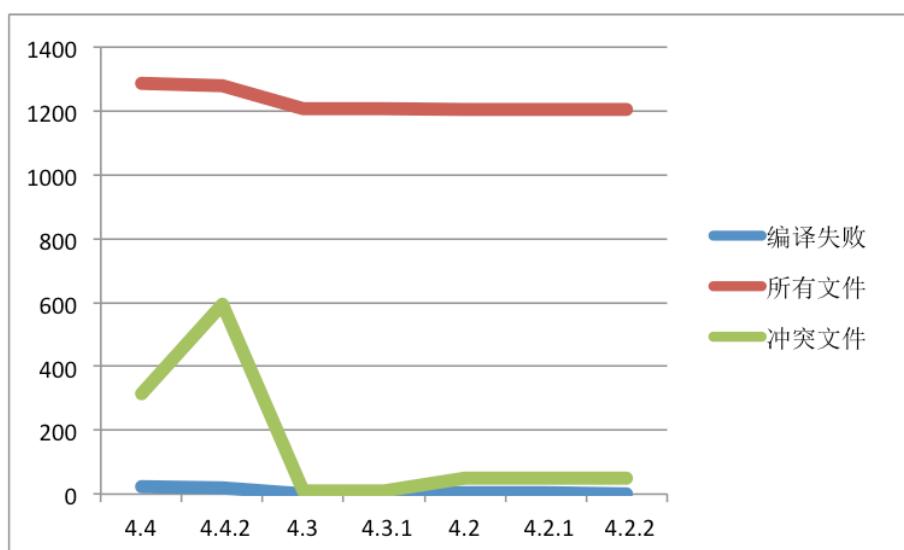


图 6.5 编译结果

如上所述，我们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对其结果进行分析。

6.3.2.1 差异性分析模块

在差异性分析模块中，我们比较关注过滤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6.6和图6.6所示。

表 6.6 差异性分析模块结果

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

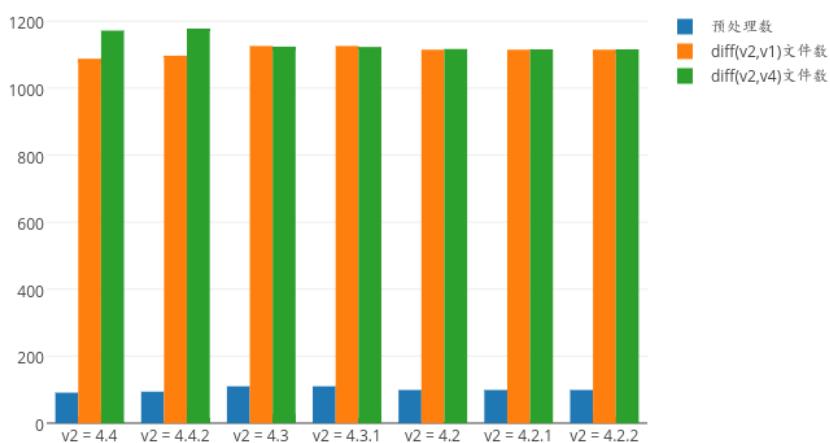


图 6.6 编译结果

可见，我们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。该算法的好处在于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6.7和表6.8所述。更直观的输出结果可以参考图??和6.8。

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

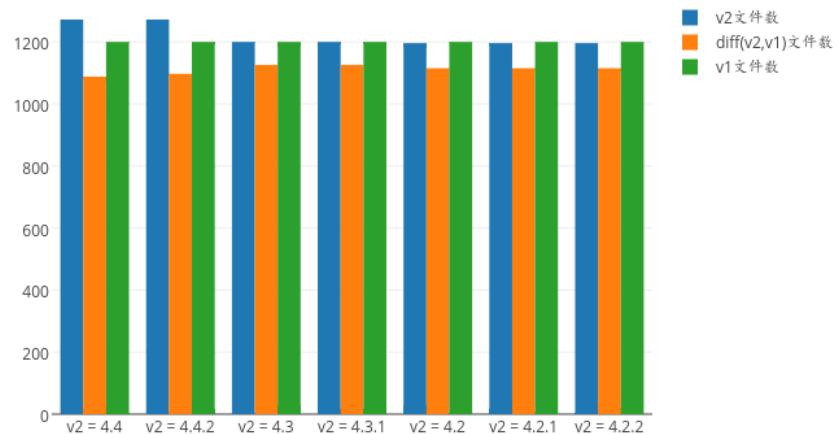


图 6.7 编译结果

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

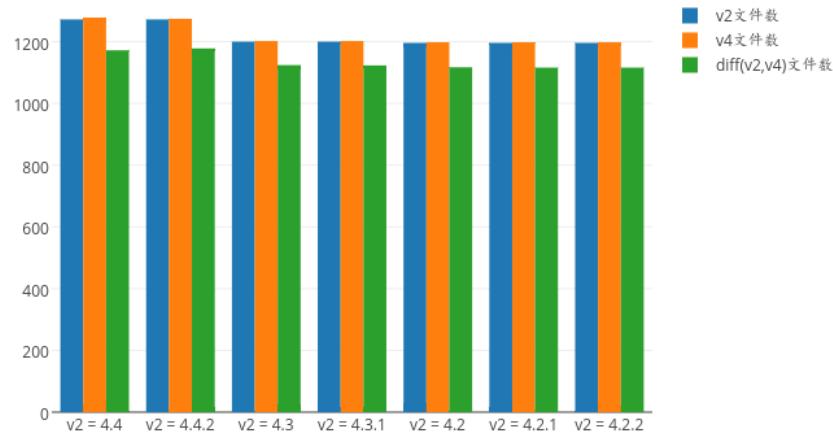


图 6.8 差异性分析模块

可见，绝大多数的文件都能够成功的进行语法差异性分析。

6.3.2.2 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6.9所述。

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。

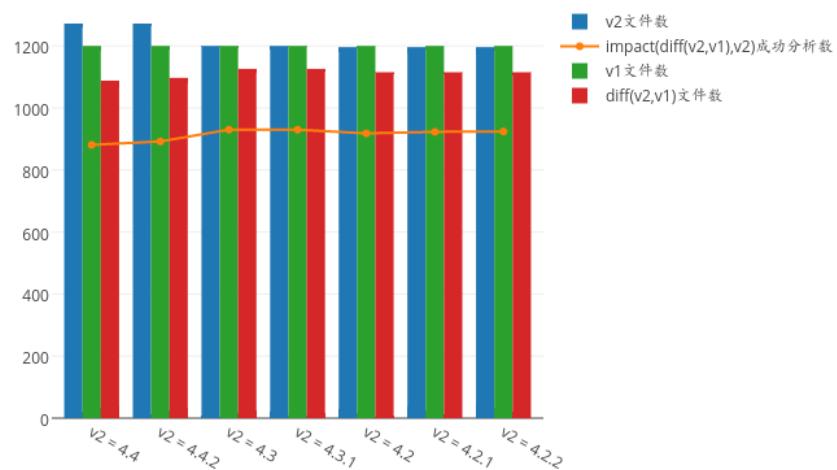


图 6.9 影响分析模块

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。

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。

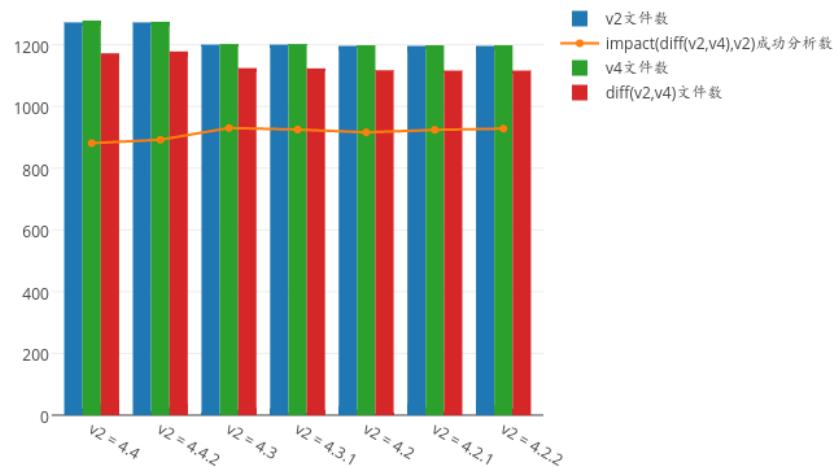


图 6.10 影响分析模块

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6.11的结果。

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。

表 6.12 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

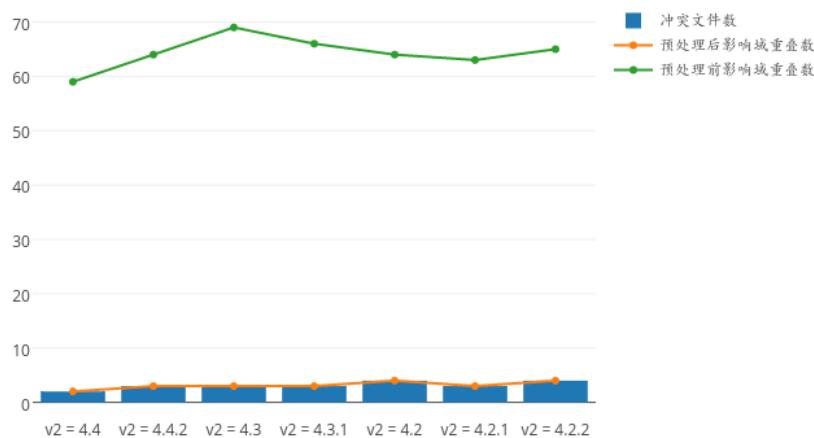


图 6.11 冲突判定模块

如图6.11所示，不使用过滤算法时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。

由于运用了过滤算法，这里找到的冲突结果都是由于针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，它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，本文中所提出的兼容性检测工具：

- 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

具体是哪种情况还有待以后做进一步的分析。虽然不排除语义冲突的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。

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。

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。

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，提出了一套补丁兼容性检测方法，它包括以下部分：

- 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：用于获取不同版本间代码的变更集合。
 - 变更语义影响分析：根据找到的变更集合，分析其变更影响域并输出。
- 软件变更冲突检测：该冲突检测方法根据得到的变更集合的影响域，找到变更间可能存在冲突的位置。

其中变更影响域分析的两个子过程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检测算法并进行了实现，更精确的分析结果目前需要人工分析的辅助，为此解决方案中需要变更语义影响分析过程提供影响追踪系统来追溯影响的来源。

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，本文的主要贡献包括：

- 对软件补丁兼容性检测的问题进行了分析。
- 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。

7.2 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] 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，他们在我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。