



北京科技大学
University of Science and Technology Beijing

密级： 公开

本科生毕业设计(论文)

题 目： 面向 BPEL 程序的变异体生成
系统重构与优化研究与实现

作 者： 王真

学 号： 41255065

学 院： 计算机与通信工程学院

专 业： 计算机科学与技术

成 绩：

2016 年 06 月

本科生毕业设计(论文)

题 目： 面向 BPEL 程序的变异体生成
 系统重构与优化研究与实现

英文题目： Refactoring and Optimization of Mutant
 Generation System for BPEL Programs

学 院： 计算机与通信工程学院

班 级： 计算机 1203

学 生： 王真

学 号： 41255065

指导教师： 孙昌爱 职称： 教授

指导教师： 职称：

声 明

本人郑重声明：所呈交的论文是本人在指导教师的指导下进行的研究工作及取得研究成果。论文在引用他人已经发表或撰写的研究成果时，已经作了明确的标识；除此之外，论文中不包括其他人已经发表或撰写的研究成果，均为独立完成。其它同志对本文所做的任何贡献均已在论文中做了明确的说明并表达了谢意。

学生签名：_____年__月__日

导师签名：_____年__月__日

毕 业 设 计（论 文）任 务 书

一、学生姓名：王真

学号：41255065

二、题目：面向 BPEL 程序的变异体生成系统重构与优化研究与实现

三、题目来源：真实 ☒ 、 自拟 ☐

四、结业方式：设计 ☐ 、 论文 ☒

五、主要内容：

在面向 BPEL 程序的变异测试理论的基础上，针对课题组研发的面向 BPEL 程序的变异体生成系统，采用设计模式对系统进行重构与优化，使系统能够生成更完备的变异体集合，支持不同方式的集成与使用，实现一个具有良好结构与可扩展的 BPEL 程序变异体生成系统。

具体任务包含以下几项：

1. 采用设计模式对课题组研发的面向 BPEL 程序的变异体生成系统进行重构与优化：前期开发的系统未考虑不同变异算子的变异体生成程序之间方法的相似性，存在代码及方法的冗余；采用设计模式重构优化系统，增强代码的可复用性，使系统结构更加清晰。
2. 增强系统生成变异体集合的完备性：目前系统变异体生成不充分，改进系统变异算子匹配及变异体生成程序，使变异体集合更加完备。
3. 提供不同方式的系统集成与使用：支持图形用户界面交互、命令行交互及插件式使用，便于与测试用例生成系统集成。
4. 设计与实现一个改进的 BPEL 程序变异体生成系统：解析测试人员输入的 BPEL 程序，根据不同变异算子的转换规则生成变异体集合。
5. 采用多个 BPEL 程序实例测试与评估系统：评估系统生成变异体集合的正确性，从变异体集合的完备性和系统的可扩展性与之前版本进行比较。

六、主要（技术）要求：

1. BPEL 服务组装语言：理解服务组装的基本原理，BPEL 的语法与编程模型，BPEL 程序中信息模块的提取技术。

2. 变异测试技术：理解变异测试技术基本过程和原理，掌握面向 BPEL 程序的变异测试技术及变异体生成方法、过程。
3. XML 文档解析技术（DOM4J）：学习 DOM4J 技术，完成对 BPEL 程序的解析。
4. 设计模式：掌握设计模式的设计原则和基本模式，采用设计模式对课题组研发的面向 BPEL 程序的变异体生成系统进行重构与优化。
5. 图形用户界面交互、命令行交互及 Eclipse 插件的开发：开发支持不同集成与使用方式的 BPEL 程序变异体生成系统，进行实例验证。

七、日程安排：

第 1-4 周：学习面向 BPEL 程序的变异测试技术，掌握 DOM4J 对 XML 文档解析的技术，撰写开题报告。

第 5-6 周：需求分析，找出系统变异体生成不充分的原因，采用设计模式对面向 BPEL 程序的变异体生成系统进行重构与优化。

第 7-10 周：学习 Java 图形用户界面、命令行交互及 Eclipse 插件开发关键技术，实现相应形式的系统，完成中期报告。

第 11-12 周：扩展与完善系统，采用多个 BPEL 程序实例测试与评估系统。

第 13-14 周：完成翻译任务，撰写论文，准备答辩。

第 15 周：答辩。

八、主要参考文献和书目：

[1]C.-A. Sun, Y. Shang, Y. Zhao, T.Y. Chen. Scenario-Oriented Testing for Web Service Compositions Using BPEL [C]. Proceedings of 12th International Conference on Quality Software (QSIC 2012), 2012:171?174.

[2]J. Vlissides, R. Helm, R. Johnson, et al. Design Patterns: Elements of Reusable Object-Oriented Software [M]. Addison -Wesley Publishing Company, 1995.

[3]G. Canfora, M.D. Penta. Service-Oriented Architectures Testing: A Survey [M]. LNCS 5413, Springer, 2009:78?105.

[4]梅彪, 姜新文, 吴恒. WS-BPEL 业务流程与访问控制 [J]. 计算机工程,

2008, 34(19) :144-146.

[5]Y. Jia, M. Harman. An Analysis and Survey of the Development of Mutation Testing [J]. IEEE Transactions on Software Engineering, 2011, 37(5):649-678.

[6]M. Bruno, G. Canfora, M.D. Penta, G. Esposito, V. Mazza. Using Test Cases as Contract to Ensure Service Compliance across Releases [C]. Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC 2005), 2005: 87-100.

[7]S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, R. Villemaire. Runtime Verification of Web Service Interface Contracts [J]. IEEE Computers, 2010, 43(3): 59-66.

[8]A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo. Mutation Operators for WS-BPEL 2.0 [C]. Proceedings of the 21th International Conference on Software & Systems Engineering and their Applications, 2008:1-7.

[9]J. Tuya, M.J. Suarez-Cabal, C. De La Riva. SQL Mutation: A tool to generate mutants of SQL database queries [C]. Proceedings of the Second Workshop on Mutation Analysis. IEEE Computer Society, 2006: 1.

[10]杨帆, 王钧玉, 孙更新. 设计模式从入门到精通 [M]. 北京: 电子工业出版社, 2010.

指导教师签字: 年 月 日

学 生 签 字: 年 月 日

系(所)负责人章: 年 月 日

摘 要

BPEL(Business Process Execution Language)是一种广泛使用的服务组装语言。随着 BPEL 技术的广泛应用,其流程定义越来越复杂,如何保证 BPEL 流程的可靠性越来越重要。变异测试作为一种基于故障的软件测试技术,是一种保证软件可靠性的重要手段。将变异测试技术应用于 BPEL 程序,可以对 BPEL 程序进行测试。

课题组前期已经研发出面向 BPEL 程序的变异体生成系统。但前期开发的系统未考虑不同变异算子的变异体生成程序之间方法的相似性,存在代码及方法的冗余,而且系统支持的变异体种类不全面,变异体生成不充分,系统的结构及可扩展性较差。

本文针对课题组研发的面向 BPEL 程序的变异体生成系统,对系统进行重构与优化、研究与实现。本文取得的主要研究成果如下:

- 1) 采用设计模式对课题组前期开发的系统进行了重构与优化:采用访问者模式、组合模式及外观模式重构优化系统,找出系统变异体生成不充分的原因,改进系统变异算子匹配及生成程序,使系统支持 34 种面向 BPEL 程序的变异算子。
- 2) 设计实现了一个改进的 BPEL 程序变异体生成系统:对系统结构重新设计和规划,开发了支持图形用户界面交互、命令行交互及 Eclipse 插件三种不同方式集成与使用的变异体生成系统。
- 3) 采用多个 BPEL 程序实例测试评估了系统生成变异体集合的正确性和完备性。

本文实现了一个具有良好结构与可扩展的 BPEL 程序变异体生成系统,增强了系统的可复用性,可扩展性及可维护性。利用本文改进的系统,测试人员可以高效快速的生成面向 BPEL 程序的变异体。

关键词: BPEL, 变异测试, 设计模式, Eclipse 插件, 软件工具

Refactoring and Optimization of Mutant Generation System for BPEL Programs

Abstract

BPEL (Business Process Execution language) is a widely used service composition language. With the wide adoption of BPEL, BPEL programs are becoming increasingly complex, and then how to enhance the reliability of BPEL programs is important. Mutation testing, a fault-based software testing technique, is a popular approach to software reliability assurance.

In previous work, we have developed a mutant generation system for BPEL programs. One major drawback is code redundancy, because the similarity among mutation operators is not taken into consideration. The other disadvantages include incomplete sets of mutation operators and mutants generated, poor system structure and extendibility.

This thesis aims to refactor and optimize the mutant generation system for BPEL programs previously developed by our group. Contributions of this thesis are as follows:

- 1) We refactored and optimized the system using visitor pattern, composite pattern and façade pattern. We also investigated the reason of incomplete generation of mutants, and improved the process of mutation operators matching and generation. The refactored system can support 34 mutation operators.
- 2) We designed and implemented an improved mutant generation system for BPEL programs. The system structure was redesigned, and the improved system can support different ways of usage, including a graphical user interface, a command line interactive and an Eclipse plug-in integration.
- 3) Multiple BPEL programs were used to evaluate the correctness and completeness of mutants generated by the refactored system.

In this thesis, a refactored and optimized mutant generation system for BPEL programs is presented, which has a good reusability, extendibility and maintainability. With such a system, testers can generate mutants for BPEL programs efficiently.

Key Words: BPEL, Mutation Testing, Design Pattern, Eclipse Plug-in, Software Tool

目 录

| | |
|---------------------------------|-----|
| 摘 要 | I |
| Abstract | III |
| 插图或附表清单 | VII |
| 1 引 言 | 1 |
| 1.1 课题背景及研究意义 | 1 |
| 1.2 国内外研究现状 | 1 |
| 1.3 研究内容与成果 | 2 |
| 1.4 论文组织结构 | 3 |
| 2 相关概念与技术 | 4 |
| 2.1 BPEL 服务组装语言 | 4 |
| 2.2 变异测试技术 | 4 |
| 2.2.1 变异测试基本思想与过程 | 4 |
| 2.2.2 面向 BPEL2.0 的变异算子 | 7 |
| 2.3 XML 文档解析技术 (DOM4J) | 8 |
| 2.4 设计模式 | 9 |
| 2.4.1 已有的设计模式 | 9 |
| 2.4.2 本课题使用的设计模式 | 10 |
| 2.5 Java 反射机制 | 12 |
| 2.6 小结 | 14 |
| 3 面向 BPEL 程序的变异体生成系统重构与优化 | 15 |
| 3.1 课题组前期工作 | 15 |
| 3.2 需求分析 | 15 |
| 3.3 总体设计 | 17 |
| 3.4 详细设计 | 17 |
| 3.5 小结 | 18 |
| 4 面向 BPEL 程序的变异体生成系统实现与演示 | 19 |
| 4.1 开发环境 | 19 |
| 4.2 工具实现 | 19 |
| 4.2.1 系统功能实现 | 19 |
| 4.2.2 系统界面实现 | 24 |
| 4.2.3 命令行交互实现 | 28 |
| 4.2.4 Eclipse 插件实现 | 30 |

| | |
|------------------------|----|
| 4.3 工具演示 | 33 |
| 4.4 小结 | 37 |
| 5 实验评估 | 38 |
| 5.1 实验目的 | 38 |
| 5.2 实验设计与过程 | 38 |
| 5.2.1 实验对象 | 38 |
| 5.2.2 实验过程 | 42 |
| 5.3 结果分析与比较 | 43 |
| 5.3.1 一阶变异体生成验证 | 43 |
| 5.3.2 二阶变异体生成验证 | 45 |
| 5.3.3 与前期开发系统的对比 | 46 |
| 5.4 小结 | 47 |
| 6 结 论 | 48 |
| 参考文献 | 49 |
| 附录 A 外文文献原文 | 51 |
| 附录 B 外文文献译文 | 73 |
| 在学取得成果 | 93 |
| 致 谢 | 95 |

插图或附表清单

| | | |
|--------|--|----|
| 图 2-1 | BPEL 程序示例 | 5 |
| 图 2-2 | 变异体示例..... | 6 |
| 图 2-3 | 传统变异测试分析流程..... | 6 |
| 图 2-4 | ASF 变异算子原理 | 7 |
| 图 2-5 | XML 示例程序 | 8 |
| 图 2-6 | 示例程序解析后形成的文档树..... | 9 |
| 图 2-7 | 外观模式结构图..... | 10 |
| 图 2-8 | 组合模式结构图..... | 11 |
| 图 2-9 | 访问者模式结构图..... | 12 |
| 图 2-10 | 抽象类 Father 与其子类 Child1、Child2 的类图..... | 13 |
| 图 2-11 | 使用 if-else 判断 | 14 |
| 图 2-12 | 使用 Java 反射机制..... | 14 |
| 图 3-1 | 系统用例图..... | 16 |
| 图 3-2 | μBPEL2.0 系统流程图 | 17 |
| 图 4-1 | 系统的包图结构..... | 19 |
| 图 4-2 | utils 包的类图 | 20 |
| 图 4-3 | parser 包的类图 | 21 |
| 图 4-4 | Visitor 模式匹配变异算子类图 | 22 |
| 图 4-5 | Visitor 模式进行变异操作类图 | 23 |
| 图 4-6 | generate 包的类图 | 24 |
| 图 4-7 | run 包的类图 | 24 |
| 图 4-8 | gui 包的包图..... | 24 |
| 图 4-9 | 系统初始进入界面..... | 25 |
| 图 4-10 | 系统使用说明界面..... | 25 |
| 图 4-11 | 系统一阶变异体生成界面..... | 26 |
| 图 4-12 | 系统二阶变异体生成界面..... | 26 |
| 图 4-13 | 系统变异算子介绍界面..... | 27 |
| 图 4-14 | 变异体展示界面..... | 27 |
| 图 4-15 | Help 窗体 | 28 |
| 图 4-16 | About 窗体 | 28 |
| 图 4-17 | CLIRun 包的类图结构..... | 29 |
| 图 4-18 | [java -jar xx.jar]命令打开 jar 包 | 29 |
| 图 4-19 | 菜单项配置结果..... | 31 |
| 图 4-20 | plugin.xml 文件内容 | 31 |
| 图 4-21 | 配置第三方 jar 包到 Classpath 面板 | 32 |
| 图 4-22 | 构建插件项目..... | 32 |
| 图 4-23 | build.properties 文件内容..... | 32 |
| 图 4-24 | SupplyChain.bpel 的一阶变异体生成..... | 33 |
| 图 4-25 | SupplyChain.bpel 生成的一阶变异体..... | 34 |

| | | |
|--------|--|----|
| 图 4-26 | SupplyChain.bpel 的二阶变异体生成 | 34 |
| 图 4-27 | SupplyChain.bpel 生成的二阶变异体 | 35 |
| 图 4-28 | 打开 uBPEL2.0.jar 包 | 35 |
| 图 4-29 | 一阶变异体生成 | 36 |
| 图 4-30 | 二阶变异体生成 | 36 |
| 图 4-31 | 插件项目集成到 Eclipse 开发平台 | 37 |
| 图 5-1 | SupplyChain 实例流程 | 39 |
| 图 5-2 | SmartShelf 实例流程 | 39 |
| 图 5-3 | LoanApproval 实例流程 | 40 |
| 图 5-4 | SupplyCustomer 实例流程 | 41 |
| 图 5-5 | TravelAgency 实例流程 | 41 |
| 图 5-6 | CarEstimate 实例流程 | 42 |
| 图 5-7 | AEL1 变异体信息 | 43 |
| 图 5-8 | 系统生成的二阶变异体类别及数量 | 46 |
| 表 2-1 | UCASE 课题组增加定义的 8 种变异算子 | 8 |
| 表 2-2 | 获取 Class 对象的方式 | 13 |
| 表 4-1 | 自定义的系统使用命令 | 30 |
| 表 5-1 | uBPEL2.0 系统与 MuBPEL 系统生成的一阶变异体情况统计 ... | 44 |
| 表 5-2 | 个别变异算子生成变异体数量不同的原因 | 45 |

1 引 言

本章介绍课题背景及研究意义、国内外研究现状、研究内容与成果，及论文的组织结构。

1.1 课题背景及研究意义

随着 Web Service 技术日益成熟和流行，许多企业都相应的创建了 Web Service 服务。由于单个 Web Service 提供的功能有限，因此需要将多个 Web Service 组合起来完成复杂的业务流程。BPEL(Business Process Execution Language)^[1]是一种基于 XML 的业务流程执行语言，可以在不改变 Web Service 正常运行的情况下，将多个 Web Service 组合到一个新的业务流程中。随着 BPEL 技术的广泛应用，其流程定义变得越来越复杂，而且由于互联网环境的开放性、Web 服务组合的松耦合性及多个 Web Service 组合协作的不稳定性，如何保证 BPEL 流程的可靠性至关重要^[1]。

变异测试^[2]作为一种基于故障的软件测试技术，是一种保证软件可靠性的重要手段。Estero-Botaro 等人提出了面向 BPEL 2.0 的 26 种变异算子^[3]，模拟程序员用工具实现 BPEL2.0 时可能出现的错误。对 Web 服务不进行修改，主要考虑 BPEL 流程描述所产生的错误类型，用另一种语法结构替换当前的语法结构，并且保证替换后程序语法的正确性，而并不保持语义的一致性^[3]。

课题组已经研发出面向 BPEL 程序的变异体生成系统^[4]。前期开发的系统未考虑不同变异算子的变异体生成程序之间方法的相似性，存在代码及方法的冗余，而且系统变异体生成不充分，系统的结构及可扩展性较差。

针对课题组研发的面向 BPEL 程序的变异体生成系统，采用设计模式对系统进行重构与优化，增强代码的可复用性，使系统结构更加清晰；改进系统变异算子匹配及变异体生成程序，使系统能够生成更完备的变异体集合；支持图形用户界面交互、命令行交互及 Eclipse 插件式使用，实现一个具有良好结构与可扩展的 BPEL 程序变异体生成系统，便于与测试用例生成系统集成。

1.2 国内外研究现状

变异测试已经被应用到各种不同的语言编写的程序当中，由于变异测试较为复杂，而且进行变异分析需要较高成本，所以研发自动化变异测试工具

十分必要^[3]。King 和 Offutt 等人^[5]定义了面向 Fortran 语言的 26 种变异算子，并将其整合到变异测试工具 Mothra 中。Ma 等人^[6]定义了面向 Java 语言的 26 种变异算子，这些变异算子被整合到面向 java 语言的变异系统 MuJava^[7,8]中，实现 java 程序的变异测试自动化，这个工具支持方法层和类层的变异。

Estero-Botaro 等人^[3]提出了面向 WS-BPEL2.0 的 26 种变异算子，模拟程序员使用 BPEL 语言开发业务流程时可能出现的典型错误。随后 Estero-Botaro 等人研发了一个面向 BPEL 程序的变异测试工具 MuBPEL^[9]，目前支持 33 种面向 BPEL 程序的变异算子，提供一阶变异体生成功能，可以对 BPEL 程序进行自动化变异测试。

课题组前期也开发了面向 BPEL 程序的变异体自动生成系统^[4]，可以支持 Estero-Botaro 等人提出的 26 种变异算子。文献[4]中详细介绍了课题组开发的自动化变异生成系统，为进一步开发自动化变异测试系统提供支持。

1.3 研究内容与成果

本课题在面向 BPEL 程序的变异测试理论的基础上，针对课题组研发的面向 BPEL 程序的变异体生成系统，采用设计模式对系统进行重构与优化，使系统能够生成更完备的变异体集合，支持不同方式的集成与使用，实现一个具有良好结构与可扩展的 BPEL 程序变异体生成系统。

主要内容如下：

- 1) 找出系统变异体生成不充分的原因，改进系统变异算子匹配及生成程序，采用设计模式对课题组研发的面向 BPEL 程序的变异体生成系统进行重构与优化。
- 2) 设计与实现一个改进的 BPEL 程序变异体生成系统，使系统支持图形用户界面交互、命令行交互及 Eclipse 插件式三种不同方式的集成与使用。
- 3) 采用多个 BPEL 程序实例测试与评估系统生成变异体集合的正确性和完备性，从变异体集合的完备性、系统代码的可复用性、系统结构及可扩展性的角度与之前版本进行比较。

预期目标如下：

- 1) 掌握面向 BPEL 程序的变异测试技术、DOM4J 解析工具对 XML 文件解析的方法及设计模式的设计原则和基本模式。
- 2) 采用设计模式重构和优化面向 BPEL 程序的变异体生成系统，使得系统代码的可复用性增强，系统结构更加清晰，生成的变异体集合更加

完备。

- 3) 改进完成的系统支持图形用户界面交互、命令行交互及 Eclipse 插件式使用。
- 4) BPEL 实例验证系统的变异体生成功能。

1.4 论文组织结构

本文的其余部分组织结构安排如下：

第 2 章介绍相关概念与技术。包括 BPEL 服务组装语言、变异测试技术、XML 文档解析技术（DOM4J）、设计模式及 Java 反射机制。

第 3 章介绍面向 BPEL 程序的变异体生成系统的重构与优化。

第 4 章介绍面向 BPEL 程序的变异体生成系统的实现与演示。

第 5 章介绍采用 BPEL 实例对系统进行实验评估。

2 相关概念与技术

本章介绍相关概念与技术，包括 BPEL 服务组装语言、变异测试技术、XML 文档解析技术（DOM4J）、设计模式及 Java 反射机制。

2.1 BPEL 服务组装语言

BPEL^[1]是一种基于 XML 的可执行服务组装语言，可以将多个 Web Service 组合成复杂的业务流程，完成更大的业务需求。BPEL 利用了若干 XML 规范^[10]：WSDL 1.1，XML Schema 1.0，Xpath 1.0 和 XSLT 1.0。BPEL 流程由 WSDL 消息和 XML Schema 类型定义提供数据模型，由 Xpath 和 XSLT 提供数据操作支持，并由 WSDL 描述 Web 服务的对外接口。BPEL 有 1.0^[11]和 2.0^[10]两个版本，本文主要研究 BPEL2.0 的内容。

BPEL 流程主要由伙伴链接、变量、故障处理程序和活动四个部分组成。BPEL 流程使用<partnerLink>定义合作伙伴链接，使用<variable>声明变量，在<faultHandlers>中定义故障处理程序。一个 BPEL 流程由多个步骤组成，每个步骤称为“活动”。BPEL 活动分为两类：基本活动和结构性活动。基本活动^[1]描述程序行为的基本步骤，包括 assign、invoke、receive、reply、throw、wait、empty、exit、rethrow 等。结构性活动^[1]描述流程的控制流逻辑，可包含其他基本活动或结构性活动，包括 scope、sequence、flow、switch、if、while、repeatuntil、foreach、pick 等。每个活动有两个可选择的标准属性：活动的 name 以及指出当连接故障发生时是否抑制的 suppressJoinFailure，同时有两个可选择的标准元素<source>和<target>，分别包括在容器<sources>和<targets>中。

图 2-1 给出了一个 BPEL 程序的示例^[3]。

2.2 变异测试技术

变异测试^[2]是一种基于故障的软件测试技术，通过将错误植入到原始程序中 come 对程序进行测试，将软件中各种潜在的故障检测出来。变异测试概念的提出最早可以追溯到 Hamlet 和 DeMillo 等人在 20 世纪 70 年代早期的研究工作^{[2][12]}。随后研究人员对该领域进行了深入研究，并做了大量的工作。

2.2.1 变异测试基本思想与过程

变异测试的可行性基于两个重要的假设：熟练程序员假设和耦合效应假

```

<flow>    ← 结构性活动
<links>    ← 容器
<link name="checkFlight-To-BookFlight"    ← 属性/>    ← 元素
</links>
<invoke name="checkFlight" ... >    ← 基本活动
<sources>    ← 容器
<source linkName="checkFlight-To-BookFlight"    ← 属性 />    ← 元素
</sources>
</invoke>
<invoke name="checkHotel" ... />
<invoke name="checkRentalCar" />
<invoke name="bookFlight" ...>
<targets>    ← 容器
<target linkName="checkFlight-To-BookFlight" />    ← 元素
</targets>
</invoke>
</flow>

```

图 2-1 BPEL 程序示例

设^[13,14]。熟练程序员假设^[13,14]即假设程序员编写的有缺陷代码与正确代码非常接近。耦合效应假设^[13,14]认为若测试用例可以检测出在原有程序上执行单一语法修改形成的缺陷即简单缺陷，那该测试用例也可以检测出在原有程序上依次执行多次单一语法修改形成的缺陷即复杂缺陷。

变异测试的基本思想^[13]是：测试人员首先根据待测程序的特征向其中植入各种类型的故障，产生大量的错误程序。产生的一个错误程序称为待测程序的一个变异体(mutant)，用来模仿某种故障的操作称为“变异算子”(mutation operator)。变异算子必须在程序中引入修改，并保持其语法的正确性。如果执行某个测试用例导致一个变异体与待测程序产生不同的结果，那么称该变异体被“杀死”(killed)，即与该变异体相关的故障能够被检测出来，反之称该变异体存活(survived)。对于任一测试用例，若在某一变异体与待测程序上的执行结果均相同，则称该变异体为等价变异体(equivalent mutants)。

图 2-2 给出了一个变异体示例。用“<”替换“>”的操作称为变异算子，将变异算子应用于待测程序，待测程序中的“>”被“<”替换，产生的错误程序称为变异体。若设计一个测试用例 a=5, b=3，则待测程序的 if 条件为真，执行结果为 c=1；变异体程序的 if 条件为假，执行结果为 c=0。变异体与待测程序的执行结果不同，表明变异体被杀死。

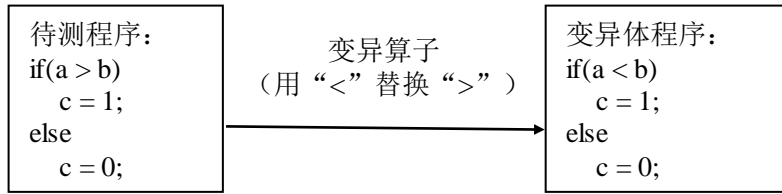


图 2-2 变异体示例

传统变异测试分析流程^[14]如图 2-3 所示。

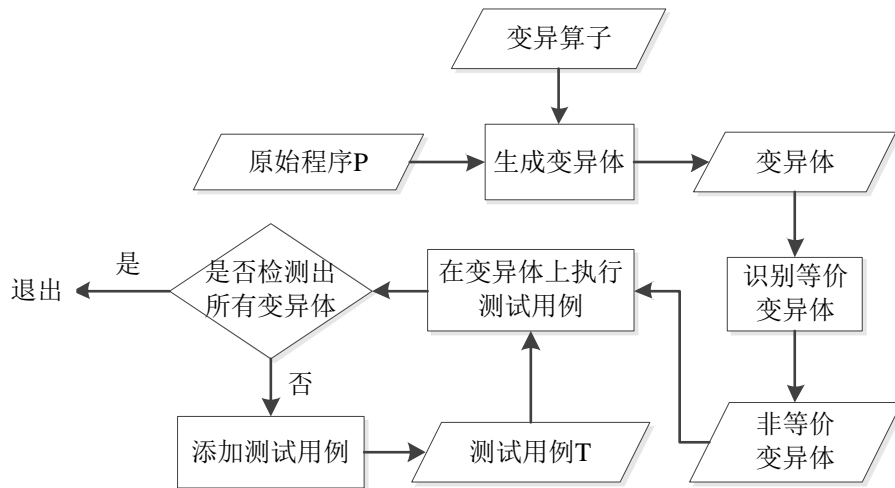


图 2-3 传统变异测试分析流程

对于给定的待测程序 P 和测试用例集 T ，定义相应变异算子，将变异算子应用于待测程序，生成大量变异体。然后识别其中的等价变异体，在得到的非等价变异体上执行测试用例集 T 中的测试用例。如果可以检测出所有的非等价变异体，即所有的非等价变异体都被杀死，则变异测试分析结束；否则说明测试用例集不完备，需要添加测试用例，直到所有的非等价变异体都被检测出。

变异测试分析可以通过变异得分（mutation score）^[14]来评估测试用例集的缺陷检测能力，变异得分为测试用例检测出的变异体数量与非等价变异体数量的比值。变异得分 $MS(P, T)$ 的计算公式如式 2-1 所示。

$$MS(P, T) = \frac{D}{(M - E)} \quad \text{式 (2-1)}$$

其中， P 为被测程序， T 为测试用例集， M 为变异体的数量， D 为被杀死的变异体数量， E 为等价变异体的数量。变异得分越高，说明测试用例集的缺陷检测能力越强。等价变异体的存在会降低变异得分，若不能检测出等价变异体，会造成巨大的资源开销。

2.2.2 面向 BPEL2.0 的变异算子

根据 BPEL 程序的语法元素类型特点, Estero-Botaro 等人^[3]提出的面向 BPEL2.0 的变异算子被分为四类, 分别用一个大写字母进行标识, 这些变异算子模拟程序员在编写 BPEL 程序时可能犯的错误。四类变异算子^[3]如下:

- 标识符替换变异算子(Identifier replacement operators) (I)
- 表达式变异算子(Expression operators) (E)
- 活动类变异算子(Activity operators) (A)
- 异常和事件变异算子(Exception and event operators) (X)

各类变异算子内部分别定义了不同操作的变异算子, 用三个大写字母进行标识, 第一个字母代表变异算子所属类别, 后两个字母标识类内的算子。

- 1) 标识符替换变异算子: 包括 ISV。该变异算子用一个相同类型、相同作用域的变量标识符替换另一个。
- 2) 表达式变异算子: 包括 EAA、EEU、ERR、ELL、ECC、ECN、EMD、EMF、EIU、EIN、EAP、EAN。例如 EEU, 用于移除所有表达式中的一元减法运算符。
- 3) 活动类变异算子: 包括与并发相关的变异算子, 即 ACI、AFP、ASF、AIS, 以及与并发无关的变异算子, 即 AEL、AIE、AWR、AJC、ASI、APM、APA。例如 APA, 用于从 pick 活动或事件处理程序中移除 onAlarm 元素。
- 4) 异常和事件变异算子: 包括 XMF、XMC、XMT、XTF、XER、XEE。例如 XEE, 用于从事件处理程序中移除 onEvent 元素。

用 ASF 变异算子举例说明。ASF 变异算子用于将 sequence 活动替换为 flow 活动。ASF 变异算子的原理如图 2-4 所示。

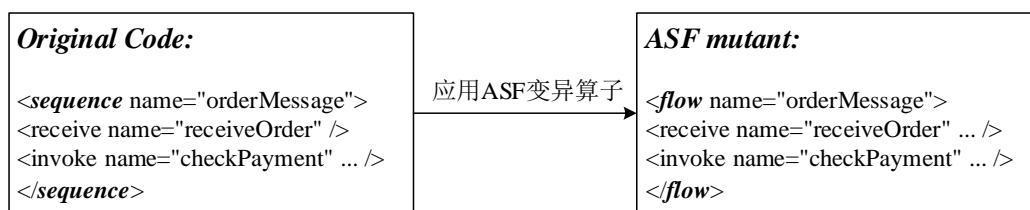


图 2-4 ASF 变异算子原理

Cádiz 大学 UCASE 课题组^[9]在研发的变异测试系统 MuBPEL 中增加定义了 8 种面向 BPEL2.0 的变异算子, 如表 2-1 所示。

表 2-1 UCASE 课题组增加定义的 8 种变异算子

| 变异算子名称 | 作用 |
|--------|--|
| EIN | Insert the XPath negation function (not) in logic expressions. |
| EIU | Insert the XPath unary minus operator in arithmetic expressions. |
| EAP | Replace a subexpression by its positive absolute value. |
| EAN | Replace a subexpression by its negative absolute value. |
| CFA | Replace an activity by an exit activity. |
| CDE | Applies the decision coverage criteria to a WS-BPEL composition. |
| CCO | Applies the condition coverage criteria to a WS-BPEL composition. |
| CDC | Applies the decision/condition coverage criteria to a WS-BPEL composition. |

2.3 XML 文档解析技术 (DOM4J)

DOM4J^[15]是一个 Java 的 XML、XPath、XSLT 开源 API，使用 Java 集合框架，支持 DOM、SAX 和 JAXP，可以用来读写 XML 文件。它具有性能优异、功能强大和易于使用的特点。DOM4J 所在包为 org.dom4j，里面定义了诸多接口，如 Attribute、Branch、Document、Element、Node 等。

DOM4J 解析器读入 XML 文档，构造一棵完整的 DOM 树形结构，通过操作树中节点，可以便捷地获取、更改、添加或删除 XML 元素。构造的 DOM 树从根部开始，并扩展到树的最底端。所有元素均可拥有子元素，相同层级上的子元素称为同胞（兄弟或姐妹）。所有元素均可拥有文本内容和属性。

图 2-5 给出了一个 XML 示例程序^[16]。

```
<bookstore>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```

图 2-5 XML 示例程序

图 2-6 给出了该程序解析后形成的文档树^[16]。根元素为 <bookstore>，

<book> 元素被包含在 <bookstore> 中。<book> 元素有 4 个子元素:<title>、<author>、<year>、<price>。

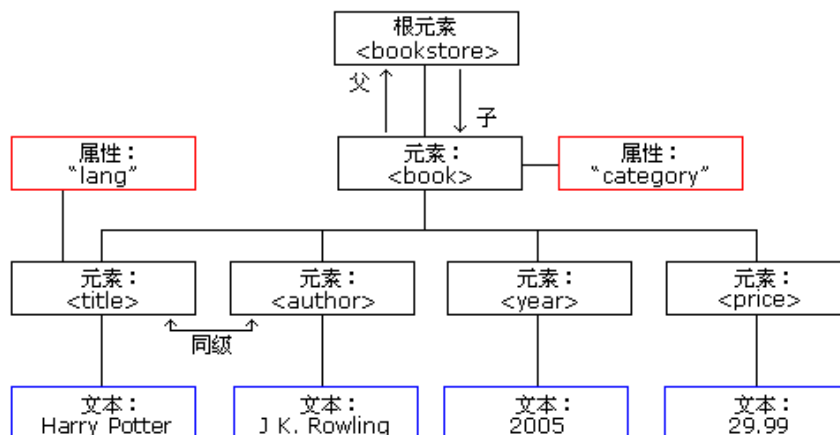


图 2-6 示例程序解析后形成的文档树

2.4 设计模式

设计模式^[17]是一套面向对象的代码设计经验总结，是在编程领域被反复使用、被多数人知晓、而且经过分类整理的代码设计方法。使用设计模式，可以提高代码的可复用性，使代码更易理解，保证代码可靠性。

2.4.1 已有的设计模式

设计模式中包含六大设计原则^[18]：单一职责原则、里氏代换原则、开放-封闭原则、依赖倒转原则、合成/聚合复用原则和迪米特法则。设计模式理论的奠基人 GoF (Gang of Four, 即 Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides)^[17]第一次将设计模式提升到理论高度，实现了设计模式的规范化。

根据文献^[17]，常用的 23 种设计模式分为三类^[17]：

1) 创建型模式：创建对象的模式，抽象了实例化的过程。

在创建对象的同时隐藏创建逻辑，而不是使用新的运算符直接实例化对象。这使程序在判断针对某个给定实例需要创建哪些对象时更加灵活。包括工厂方法模式、抽象工厂模式、建造者模式、原型模式、单例模式。

2) 结构型模式：关注类和对象的组合。

继承的概念被用来组合接口和定义组合对象获得新功能的方式。包括适配器模式、桥接模式、代理模式、外观模式、装饰模式、组合模式、享元模

式。

3) 行为型模式：描述了对对象和类的模式，以及它们之间的通信模式，可分为行为类模式和行为对象模式。

行为类模式使用继承机制在类间分派行为，行为对象模式使用对象聚合来分配行为。包括命令模式、观察者模式、责任链模式、迭代器模式、访问者模式、状态模式、备忘录模式、策略模式、调停者模式、模版方法模式、解释器模式。

2.4.2 本课题使用的设计模式

在本课题的研究中，主要使用外观模式、组合模式和访问者模式对系统进行重构。

1) 外观（Facade）模式

外观模式^[17]表示为子系统的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

外观模式有如下优点：

- 实现了子系统与客户端之间的松耦合关系。
- 客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使得子系统使用起来更加容易。

外观模式的结构图^[18]如图 2-7 所示。

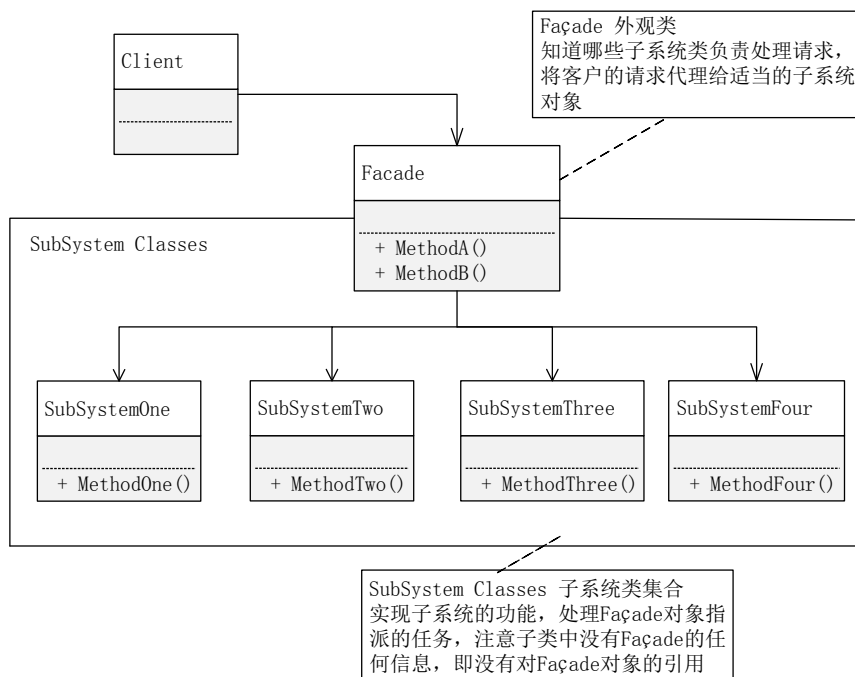


图 2-7 外观模式结构图

2) 组合（Composite）模式

组合模式^[17]表示将对象组合成树形结构以表示“部分整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

在树型结构的问题中使用组合模式，可以模糊简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

组合模式的结构图^[18]如图 2-8 所示。

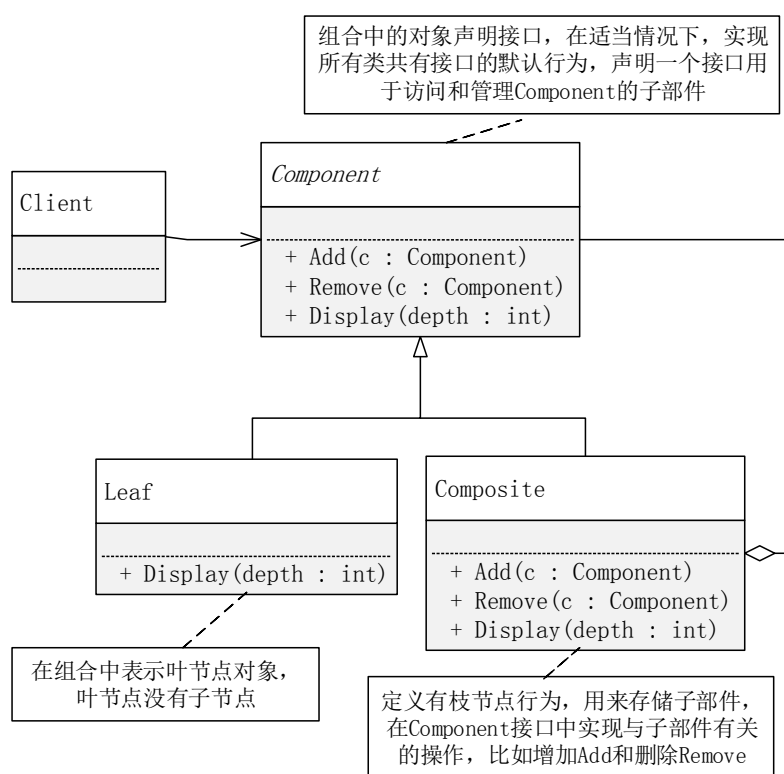


图 2-8 组合模式结构图

3) 访问者（Visitor）模式

访问者模式^[17]表示一个作用于某对象结构中的各元素的操作，可以在不改变各元素类的前提下定义作用于这些元素的新操作。访问者模式的根本目的是将对数据的处理从数据结构中分离出来。适合数据结构相对稳定的系统。

访问者模式的结构图^[18]如图 2-9 所示。

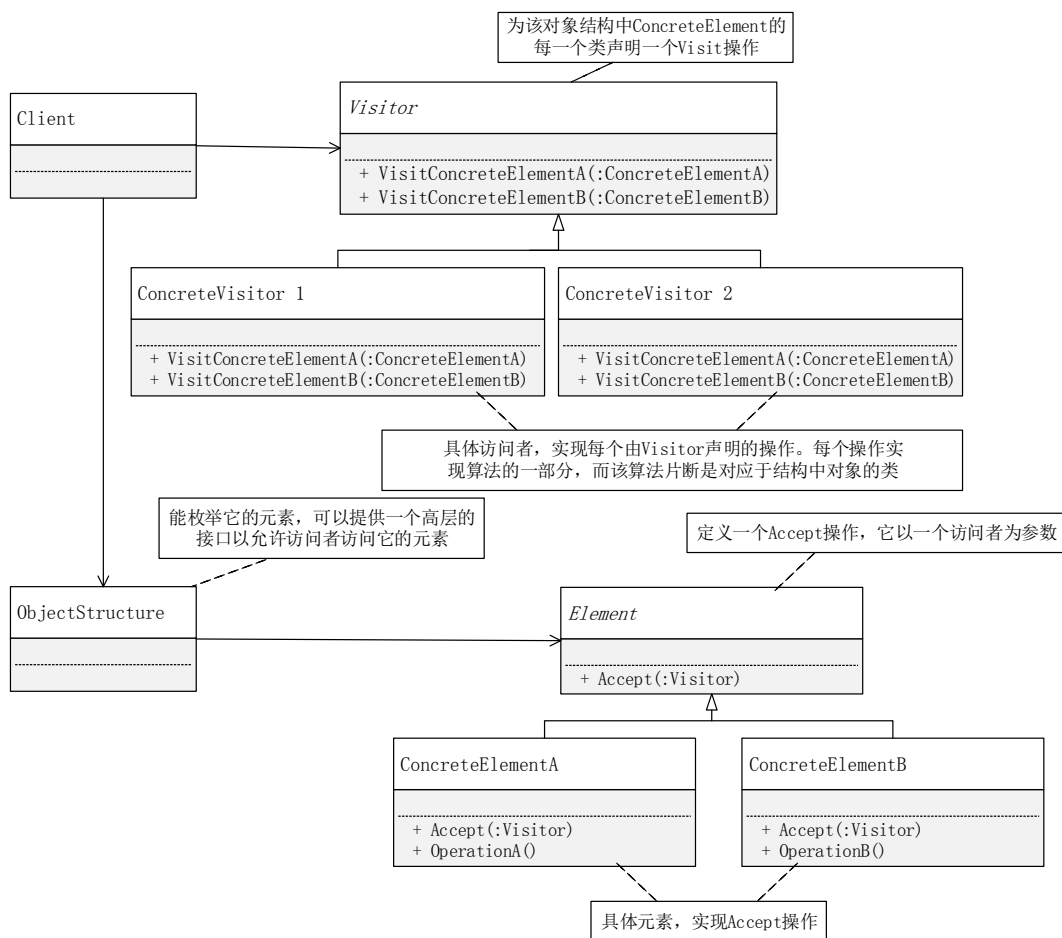


图 2-9 访问者模式结构图

2.5 Java 反射机制

在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制^[19]。

在 JDK 中，主要由以下类来实现 Java 反射机制，这些类都位于 java.lang.reflect 包中：

- Class 类：代表一个类（这个类很特殊，位于 java.lang 包下）。
- Field 类：代表类的成员变量（成员变量也称为类的属性）。
- Method 类：代表类的方法。
- Constructor 类：代表类的构造方法。
- Array 类：提供了动态创建数组，以及访问数组的元素的静态方法。

可采用多种方式获取 Class 对象，表 2-2 列出了获取 Class 对象的方式^[20]。

表 2-2 获取 Class 对象的方式

| 获取方式 | 说明 | 示例 |
|--------------------------------|-------------------------|---|
| object.getClass() 每个对象都有此方法 | 获取指定实例对象的 Class | List list = new ArrayList(); Class listClass = list.getClass(); |
| class.getSuperclass() | 获取当前 Class 的继承类 Class | List list = new ArrayList(); Class listClass = list.getClass(); Class superClass = listClass.getSuperclass(); |
| Object.class | .class 直接获取 | Class listClass = ArrayList.class; |
| Class.forName (类名) | 用 Class 的静态方法, 传入类的全称即可 | try { Class c=Class.forName("java.util.ArrayList"); } catch (ClassNotFoundException e) { e.printStackTrace(); } |
| Primitive.TYPE | 基本数据类型的封装类获取 Class 的方式 | Class longClass = Long.TYPE; Class integerClass = Integer.TYPE; Class voidClass = Void.TYPE; |

实例化无参构造函数的对象, 有以下两种方式:

- 1) Class.newInstance();
- 2) Class.getConstructor(new Class[]{}).newInstance(new Object[]{})

实例化带参构造函数的对象:

```
clazz.getConstructor(Class<?>... parameterTypes).  
newInstance(Object... initargs)
```

下面给出一个示例说明使用 Java 反射机制的优点。图 2-10 所示类图表示抽象类 Father 与两个子类 Child1、Child2 的关系。通过比较图 2-11 和图 2-12 实例化 Child1、Child2 子类, 并调用其 run() 方法的方式, 可以看出 Java 反射机制的使用可以避免 if 和 switch 使用的缺点, 使得程序的灵活性大幅度提高。同时, 也是父类和子类之间多态性的一种体现。

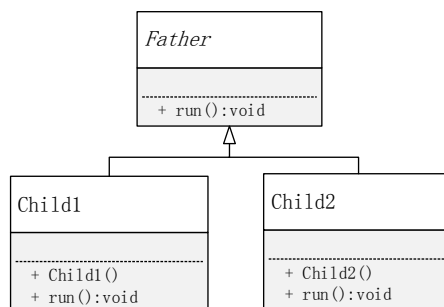


图 2-10 抽象类 Father 与其子类 Child1、Child2 的类图

```
String[] s = {"Child1", "Child2"};
for(int i = 0; i < s.length; i++)
{
    if(s[i].equals("Child1")){
        Child1 child1 = new Child1();
        child1.run();
    }
    else if(s[i].equals("Child2")){
        Child2 child2 = new Child2();
        child2.run();
    }
    ...
}
```

图 2-11 使用 if-else 判断

```
Father cc;
String[] s = {"Child1", "Child2"};
for(int i = 0; i < s.length; i++){
    try {
        Class<?> c = Class.forName(s[i]);
        try {
            cc = (Father) c.newInstance();
            cc.run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

图 2-12 使用 Java 反射机制

2.6 小结

本章介绍了与本文研究内容相关的概念与技术，包括 BPEL 服务组装语言，变异测试技术的基本思想与过程、面向 BPEL2.0 的变异算子，DOM4J 解析 XML 文档的原理，设计模式的分类及本文使用的外观模式、组合模式、访问者模式的概念及结构，使用 Java 反射机制的优点。初步了解面向 BPEL 程序的变异测试过程及设计模式的使用方法。

3 面向 BPEL 程序的变异体生成系统重构与优化

本章首先讨论课题组前期工作的不足，然后对系统进行重构与优化，包括对系统进行需求分析、总体设计和详细设计。

3.1 课题组前期工作

课题组前期已研发出面向 BPEL 程序的变异体生成工具^[4]，能够根据一组变异算子自动生成面向 BPEL 程序的变异体。但存在一些不足之处：

- 1) 未考虑不同变异算子的变异体生成程序之间方法的相似性，存在代码及方法的冗余。部分变异算子的变异操作基本类似，如 ACI、AFP、AIS 均是用于改变属性值，在程序编写时应进行归类和抽象，提高代码的复用性。
- 2) 变异体种类不全面，生成的变异体数量不充分。目前只支持 Estero-Botaro 等人^[3]提出的 26 种变异算子，而且部分变异体生成程序生成变异体不完备。
- 3) 系统结构不够清晰，系统可扩展性、可复用性及可维护性较差。若添加或修改变异算子，添加高阶变异体生成功能，系统需进行大幅度修改，代码修改及维护困难，扩展性较差。

3.2 需求分析

本文针对前期研发的系统存在的问题，对系统进行改进，并扩展前期研发的系统功能。具体需求如下：

- 1) 采用设计模式对系统进行重构与优化，使系统的结构更加清晰，增强系统的可扩展性和可维护性。
- 2) 对变异操作类似的变异算子进行归类和抽象，减少代码冗余，提高代码的复用性。
- 3) 增加变异算子种类，使变异体集合更加完备。
- 4) 扩展系统功能，增加二阶变异体生成功能，同时使系统支持图形用户界面、命令行交互及 Eclipse 插件三种不同形式的集成和使用。

系统用例图如图 3-1 所示。

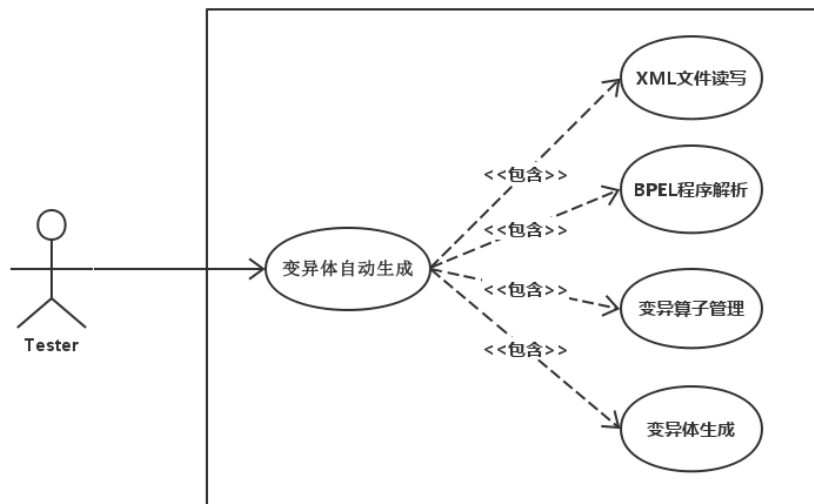


图 3-1 系统用例图

各用例具体介绍如下：

变异体自动生成用例：测试人员提供待测 BPEL 程序，根据测试人员的变异类型及变异体阶数需求，应用变异算子，产生面向 BPEL 程序的变异体。包括四个子用例：XML 文件读写用例、BPEL 程序解析用例、变异算子管理用例及变异体生成用例。

- **XML 文件读写用例：**提供读写 XML 文件的接口。通过 DOM4J XML 解析工具，读入待测 BPEL 文件，将其解析为 DOM 树，返回解析后的 document 对象，便于对其进行修改、查找等操作。在应用变异算子对待测 BPEL 程序植入错误后，运用 DOM4J 解析工具将生成的变异体文件写出。
- **BPEL 程序解析用例：**采用访问者模式，通过变异算子管理用例，查找可与变异算子匹配的节点，并记录相应的变异算子名称及节点位置。
- **变异算子管理用例：**定义变异算子匹配及处理操作。采用访问者模式和组合模式构建组织各变异算子类。提供匹配 BPEL 程序可变异的节点，应用变异算子植入错误的功能。
- **变异体生成用例：**该用例根据 BPEL 程序解析得到的可变异节点信息，根据用户需求，通过变异算子管理用例，对可变异节点进行变异操作，生成一阶或二阶变异体，采用外观模式提供生成变异体的统一接口。

3.3 总体设计

根据系统需求, 本文对原有系统进行重构和优化, 设计了面向 BPEL 程序的变异体生成系统 μ BPEL2.0。 μ BPEL2.0 的系统流程图如图 3-2 所示。

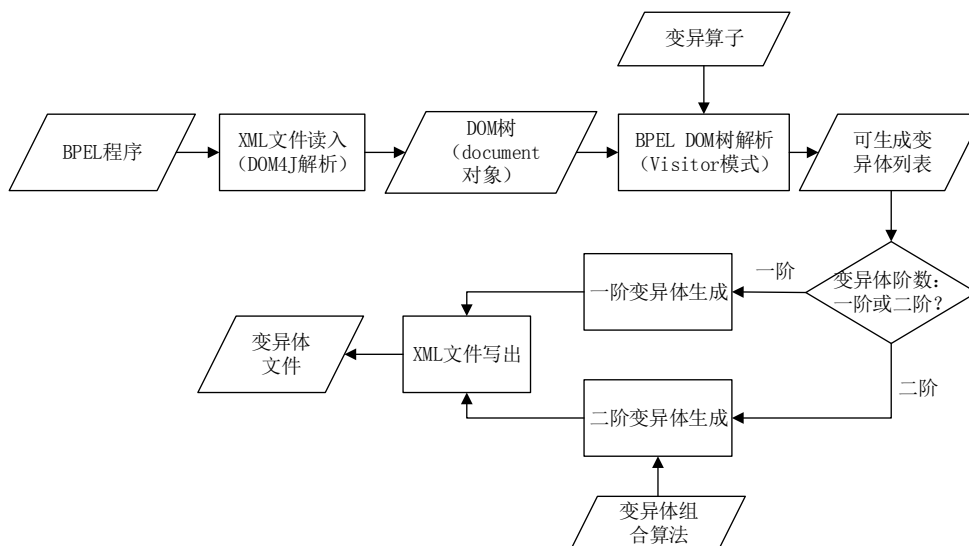


图 3-2 μ BPEL2.0 系统流程图

给定一个待测 BPEL 程序, 通过 DOM4J 解析工具解析, 得到待测程序的树形 document 对象, 将预先定义好的变异算子应用于该 BPEL 程序, 通过解析 document 对象, 获得该 BPEL 程序可以变异位置的信息。最终可以选择生成的变异体阶数, 并以文件的形式输出生成的变异体。

3.4 详细设计

根据系统需求及系统流程图, 将 μ BPEL2.0 分为四个功能模块: XML 文件读写模块、BPEL 程序解析模块、变异算子管理模块及变异体生成模块。各模块具体功能如下:

1) XML 文件读写模块

负责读入和写出 BPEL 程序文件。BPEL 文件以 XML 形式实现, 通过 DOM4J XML 文件解析工具读入 BPEL 文件, 将其解析为 DOM 树形结构的 document 对象, 便于对其进行查找和修改。document 树中的节点 (Node) 类型可分为元素 (Element)、属性 (Attribute)、命名空间 (Namespace)、文本 (Text) 等。同时, 通过继承并重写 DOM4J 解析工具自带的 XMLWriter 类, 可以以基于 XML 的格式写出生成的变异体文件。

2) BPEL 程序解析模块

负责解析 document 对象，获取待测 BPEL 程序可以变异位置的信息。根据需要执行的变异算子列表，遍历 document 对象，查找可变异的节点，并记录节点位置。为对原有系统进行优化，该模块采用访问者模式和组合模式实现。

通过继承 DOM4J 自带的 Visitor 接口，定义访问 document 对象不同类型节点的操作方法，从而优化变异算子与 document 对象的匹配过程，增强系统的可扩展性。若需添加新的变异算子，只需添加相应变异算子类，继承 Visitor 接口，实现访问相应节点的操作方法即可。同时，采用组合模式组织各类型的节点类，系统的层次关系更加明确，系统的可维护性较好。

3) 变异算子管理模块

定义各变异算子的匹配与操作处理方法。该模块是系统的核心，每个变异算子均定义两个相关类，一个负责匹配 BPEL 程序，获取可以变异的节点，一个负责对相应节点进行变异处理，并将变异体写出。

该模块采用访问者模式进行实现，变异算子匹配类继承 DOM4J 的 Visitor 接口，实现访问相应节点的操作方法；变异操作处理类继承 DOM4J 的 XMLWriter 类，并重写相应变异节点的 write 方法。该模块在前期开发系统的基础上，增加实现 8 个新的变异算子，用于提高变异体集合的完备性。同时，对变异操作相似的变异算子进行归类和抽象，明确各类之间的层次关系，减少代码冗余，提高代码的复用性。

4) 变异体生成模块

负责生成一阶和二阶变异体。根据记录的可变异位置信息，调用相应变异算子处理方法，生成一阶或二阶变异体，并将变异体文件写出。

该模块采用外观模式实现，通过一个外观类，将请求传递给一阶或二阶变异体生成子系统，实现客户端和子系统之间的松耦合。

3.5 小结

本章主要讨论了课题组前期工作存在的不足和改进的面向 BPEL 程序的变异体生成系统的设计，并指出了为解决课题组前期开发的系统存在的不足所做的工作。

4 面向 BPEL 程序的变异体生成系统实现与演示

本章介绍系统的开发环境、工具实现，包括系统功能实现、界面实现、命令行交互实现及 Eclipse 插件实现，并运用一个实例进行工具的演示。

4.1 开发环境

该系统使用 Java 语言，运用 Eclipse 和 NetBeans 开发工具进行开发。开发所用操作系统为 64 位 Win7 旗舰版操作系统，处理器为主频 2.6GHZ、双核四线程的 Inter Core-i5 处理器，系统内存为 4GB。

4.2 工具实现

本文使用 Java 语言进行系统开发，最终实现一个代码量约为 1 万行的改进的面向 BPEL 程序的变异体生成系统。下面从系统功能实现、系统界面实现、命令行交互实现及 Eclipse 插件实现四个方面具体介绍系统实现。

4.2.1 系统功能实现

根据系统的模块划分，系统功能部分的包图设计如图 4-1 所示。

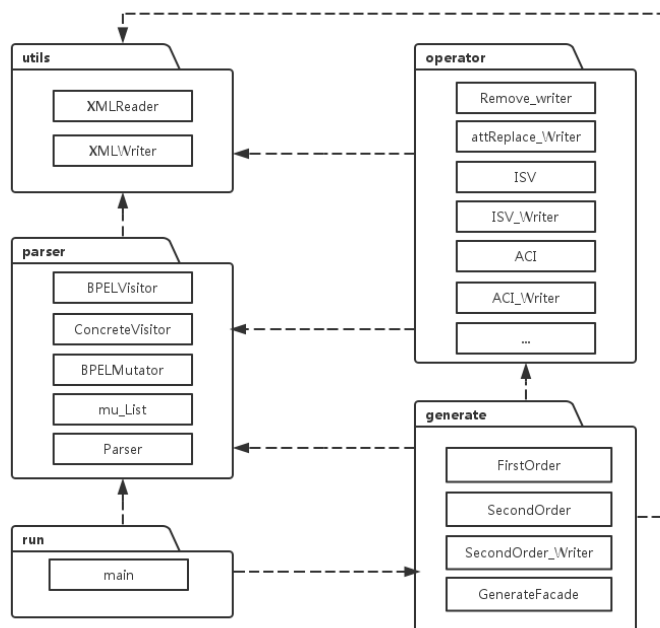


图 4-1 系统的包图结构

包图结构具体介绍如下：

1) utils 包

包含两个类：XMLReader 类和 XMLWriter 类。用于读写 XML 文件。相应类图如图 4-2 所示。

各类的功能如下：

- XMLReader 类：读入 BPEL 源程序，经 DOM4J 解析，返回 BPEL 源程序对应的 document 对象（DOM 树）。
- XMLWriter 类：改写 DOM4J 解析工具自带的 XMLWriter 类，用于生成的变异体文件的写出。

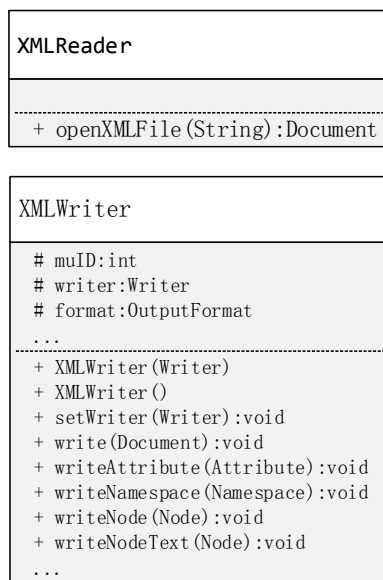


图 4-2 utils 包的类图

2) parser 包

包含 BPELVisitor 类、ConcreteVisitor 类、BPELMutator 类、mu_List 类和 Parser 类。用于解析 BPEL 文件。相应类图如图 4-3 所示。

各类的功能如下：

- BPELVisitor 类、ConcreteVisitor 类、BPELMutator 类：继承自 DOM4J 解析工具的 Visitor 接口，构成 Visitor 模式，定义访问 Document、Element、Attribute 等的方法。BPELMutator 类主要定义和变异体有关的方法，主要方法及其功能如下：

getMutantsList(): 获取记录可变异位置信息的列表。

getMutantID(): 获取变异体的 ID。

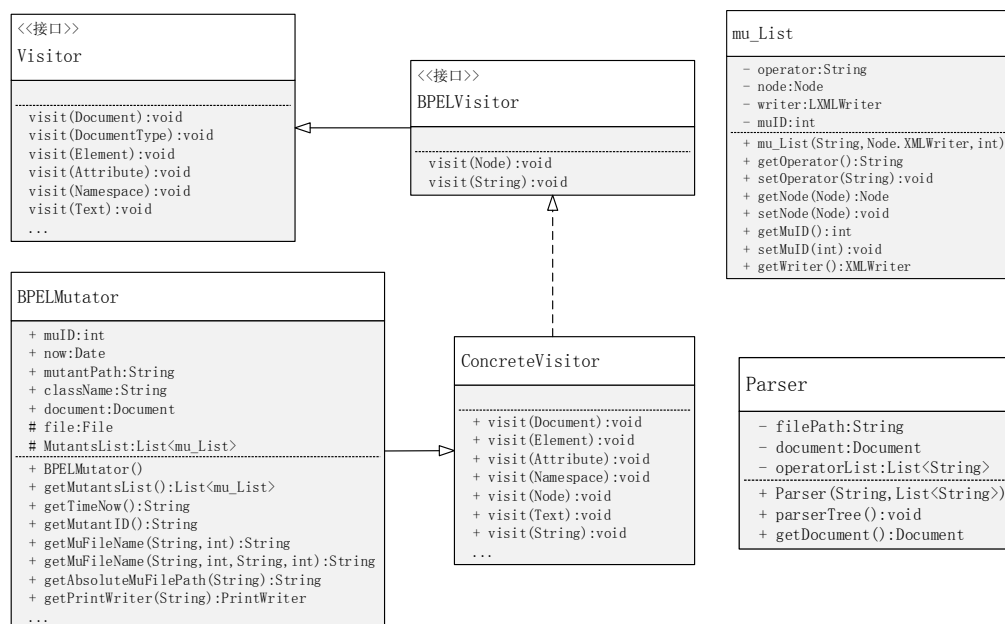


图 4-3 parser 包的类图

getMuFileName(String, int, String, int): 根据变异算子名称及 ID 获取二阶变异体文件的名称。

getAbsolutePathMuFilePath(String): 根据变异体文件的名称获取变异体文件的绝对路径。

- **mu_List 类:** 定义用于存储可变异位置信息的数据结构。包括变异算子名称、变异体 ID、变异节点及 writer 对象。
- **Parser 类:** 用于解析 BPEL 文件的 document 对象，获取可变异位置信息列表。

3) operator 包

定义变异算子匹配及处理方法，是该变异体生成系统中最关键的部分。

根据变异算子转换规则的特点，34 个变异算子可以分为四类，具体如下：

- **属性值替换类:** ACI、AFP、AIS、XTF 变异算子。该类变异算子植入属性值改变的错误，分别改变 createInstance、parallel、isolated、fault name 属性的值，因此可以将其操作抽象为 attReplace_Writer 类，该类变异算子的 Writer 类均继承 attReplace_Writer 类。
- **表达式替换类:** CCO、CDC、CDE、EAA、EAN、EAP、ECC、ECN、EEU、EIN、EIU、ELL、EMD、EMF、ERR 变异算子。该类变异算子均植入与表达式替换有关的错误。如 EAA 替换算术运算符，ERR 替换关系运算符，ELL 替换逻辑运算符。该类变异算子的操作抽象

为 `exReplace_Writer` 类，该类变异算子的 `Writer` 类均继承 `exReplace_Writer` 类。

- 移除节点类：AEL、AIE、AJC、APA、APM、XEE、XER、XMC、XMF、XMT 变异算子。该类变异算子植入移除节点的错误，将其操作抽象为 `Remove_Writer` 类，该类变异算子的 `Writer` 类均继承 `Remove_Writer` 类。
- 其他：ASF、ASI、AWR、CFA、ISV 变异算子。该类变异算子的转换规则和其他变异算子无相似之处，因此其 `Writer` 类单独定义，直接继承 `XMLWriter` 类，重写 `write` 方法。

将转换规则相似变异算子的操作进行抽象，提高了代码的可复用性，同时减少了代码冗余，改善了代码结构。

根据变异算子的分类及功能需求，该包主要包含两大类，变异算子类和变异算子 `Writer` 类。各类的功能如下：

- 变异算子类：如 ACI、AEL、AFP 等。继承 `BPELMutator` 类，重写 `ConcreteVisitor` 中的 `visit` 方法，进行 document 树的遍历查找。采用 `Visitor` 模式匹配变异算子，类图结构如图 4-4 所示。

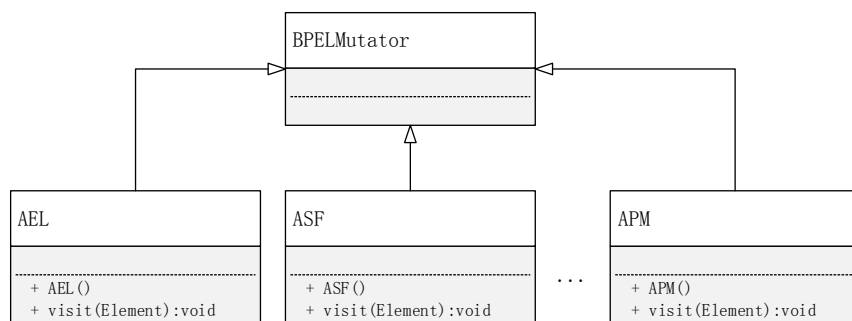


图 4-4 Visitor 模式匹配变异算子类图

- 变异算子 `Writer` 类：如 `ASF_Writer`、`Remove_writer`、`attReplace_Writer` 等。继承 `XMLWriter` 类。重写 `XMLWriter` 中的 `write` 方法，进行变异体文件的写出。同样采用 `Visitor` 模式进行变异操作，类图结构如图 4-5 所示。

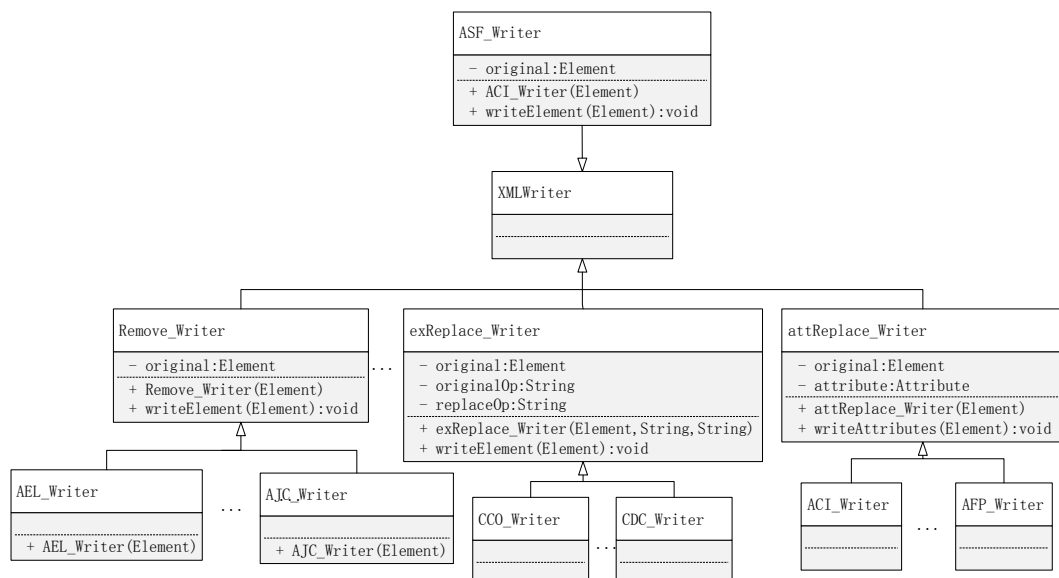


图 4-5 Visitor 模式进行变异操作类图

4) generate 包

包含 FirstOrder、SecondOrder、SecondOrder_Writer、GenerateFacade 类。用于产生一阶和二阶变异体。采用外观模式构造，类图结构如图 4-6 所示。

各类功能如下：

- FirstOrder 类：生成一阶变异体，并记录生成的变异体信息。
- SecondOrder 类：生成二阶变异体，并记录生成的变异体信息。
- SecondOrder_Writer 类：继承 XMLWriter 类，重写 write 方法，复用各变异算子 Writer 类中的 write 方法，负责写出生成的二阶变异体文件。
- GenerateFacade 类：外观类，提供调用一阶和二阶变异体生成子系统的接口，减少客户端和变异体生成子系统之间的耦合。

5) run 包

包含 main 类。提供使用该变异体生成系统的接口，可以隐藏系统内部的实现细节。类图结构如图 4-7 所示。传入待执行的变异算子列表，BPEL 文件路径，及生成一阶或二阶变异体的标识，便可根据客户需求生成需要的变异体。

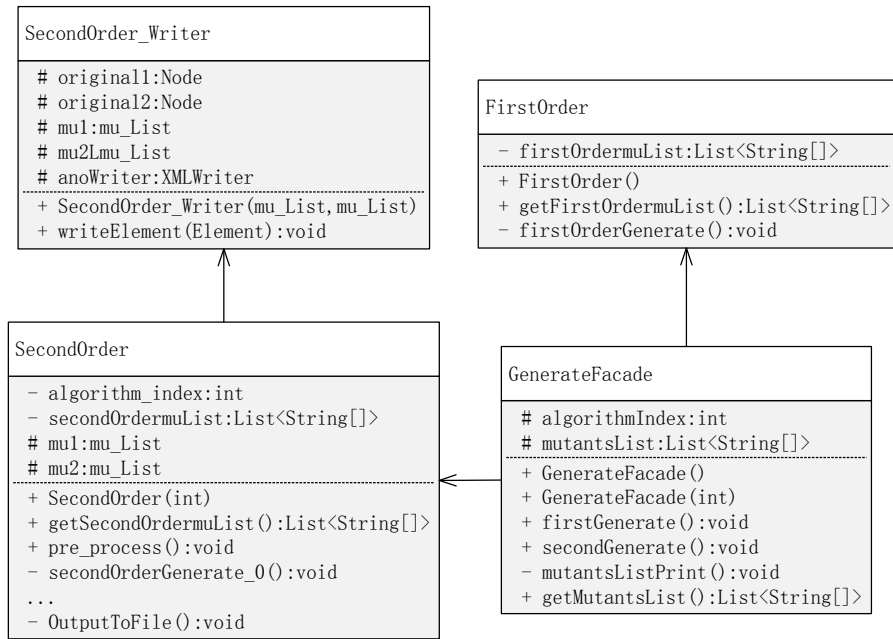


图 4-6 generate 包的类图

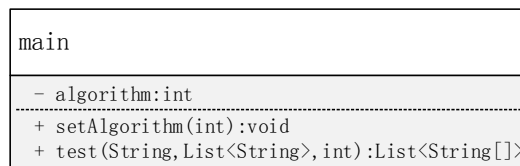


图 4-7 run 包的类图

4.2.2 系统界面实现

系统界面设计部分在项目的 gui 包中，包的结构如图 4-8 所示。

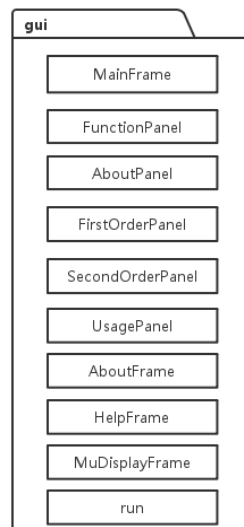


图 4-8 gui 包的包图

根据系统的模块划分，系统界面设计为一个主界面，四个功能界面，一个变异体展示界面。同时设计对应菜单项的两个窗体。各界面具体设计如下：
图 4-9 是系统的初始进入界面。点击按钮 Start 可进入系统。

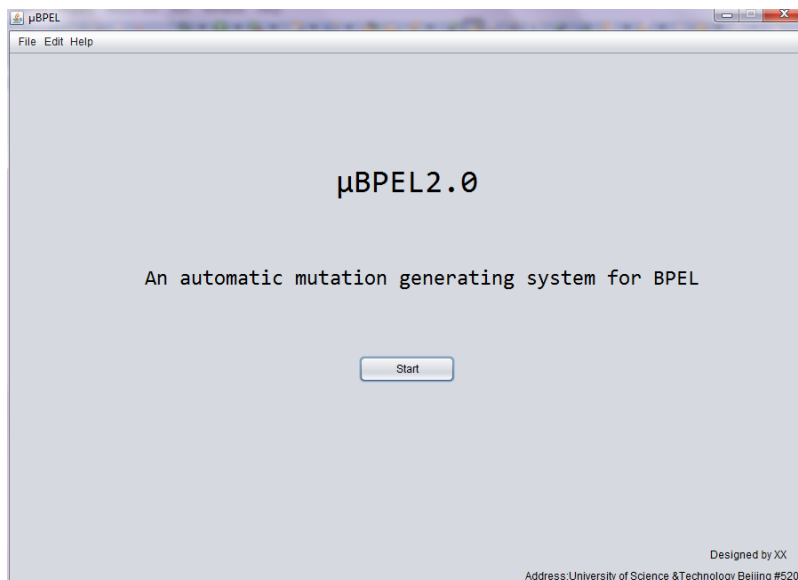


图 4-9 系统初始进入界面

图 4-10 是系统功能界面中的系统使用说明界面。该界面帮助用户了解使用该系统生成一阶和二阶变异体的步骤。

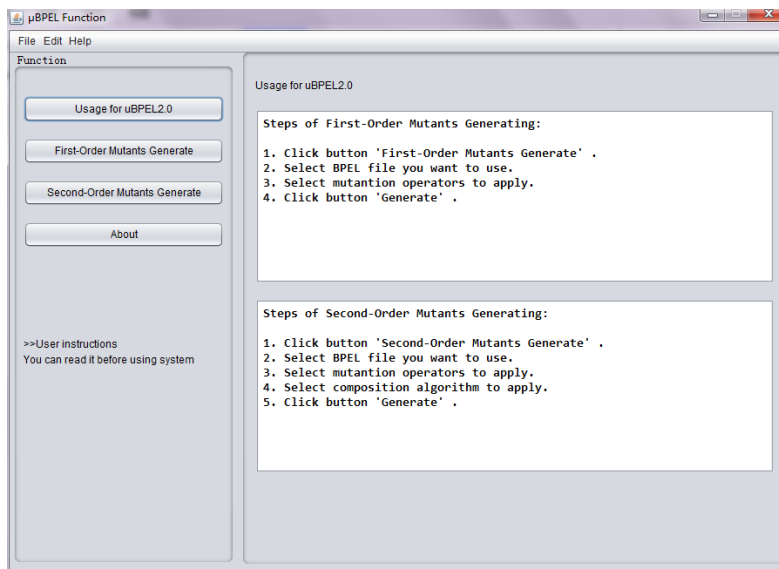


图 4-10 系统使用说明界面

图 4-11 是系统一阶变异体生成界面。选择待测的 BPEL 程序，同时选择需要执行的变异算子，系统便可根据用户需求生成需要的一阶变异体。

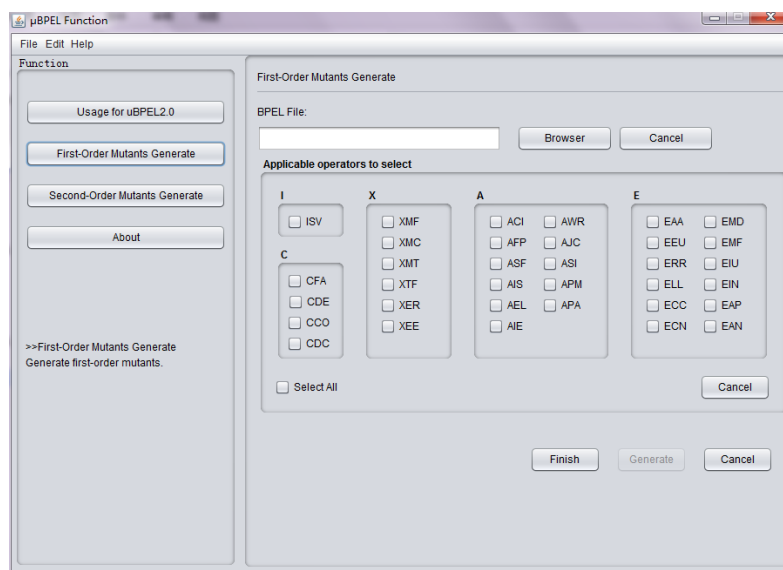


图 4-11 系统一阶变异体生成界面

图 4-12 是系统二阶变异体生成界面。用户需要选择待测 BPEL 程序和需要执行的变异算子，同时需要选择变异体组合算法确定一阶变异体的组合方式。

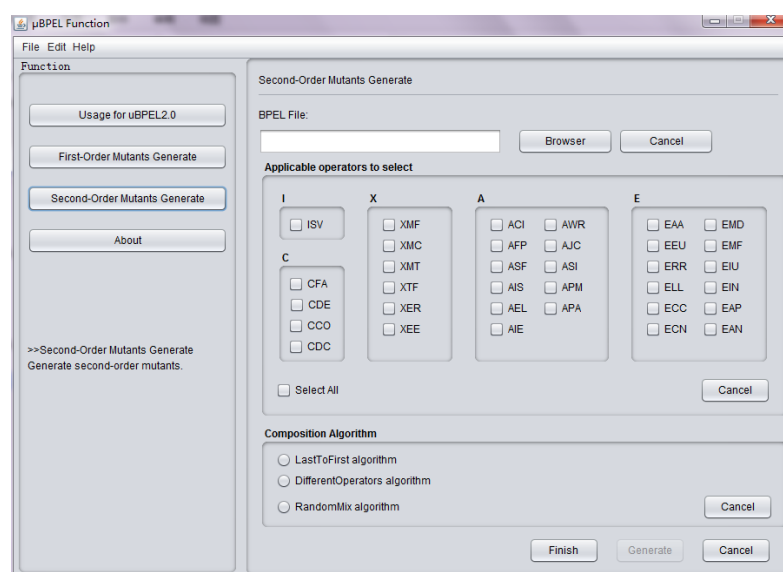


图 4-12 系统二阶变异体生成界面

图 4-13 是系统的变异算子介绍界面。该界面介绍 34 个变异算子的错误植入规则，以使用户更好的理解生成的变异体文件。

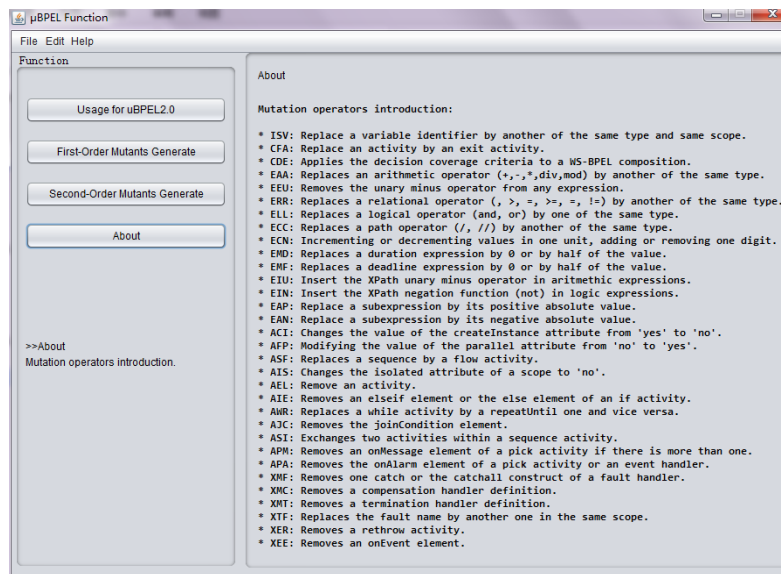


图 4-13 系统变异算子介绍界面

图 4-14 是变异体展示界面。该界面展示系统生成的变异体列表及数量，在右侧上方区域显示待测 BPEL 程序的内容，右侧下方显示生成的变异体文件的内容。

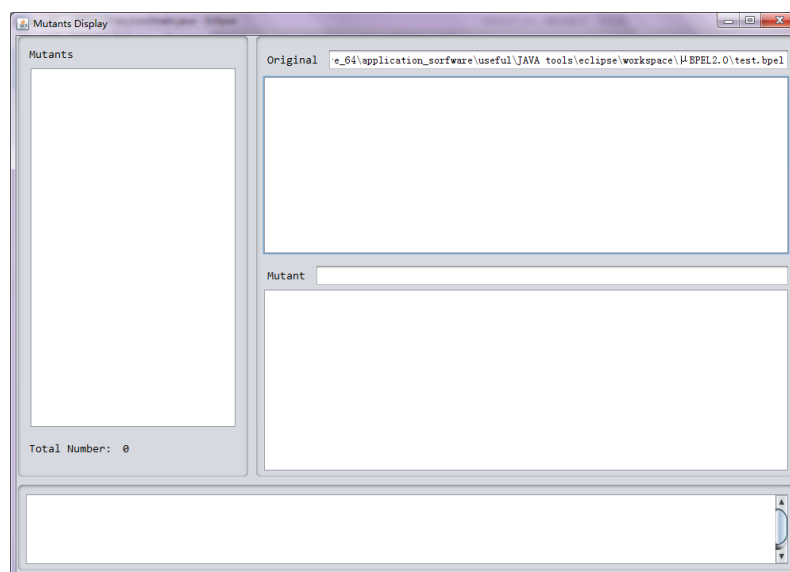


图 4-14 变异体展示界面

图 4-15 和图 4-16 是系统菜单项的 Help 窗体和 About 窗体，为用户在系统使用过程中提供指导。

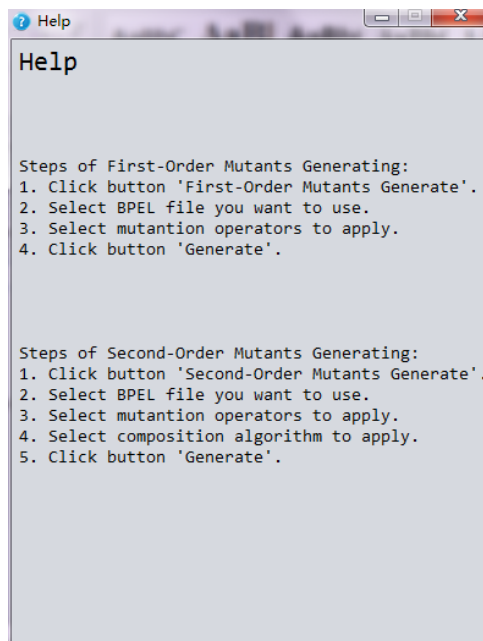


图 4-15 Help 窗体

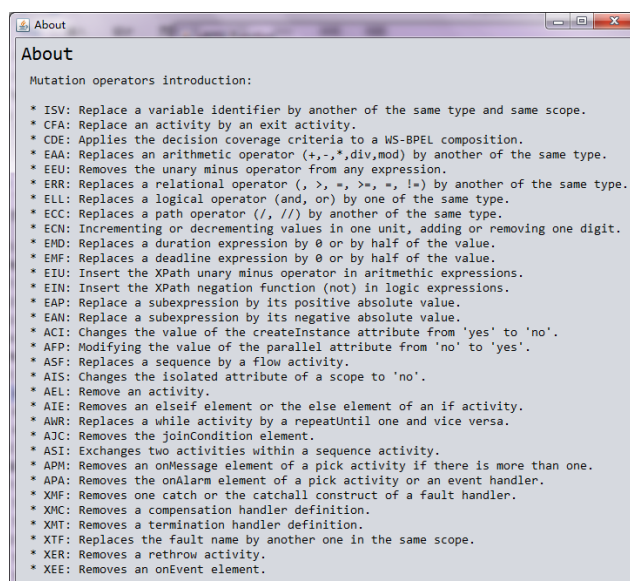


图 4-16 About 窗体

4.2.3 命令行交互实现

系统命令行交互实现在项目的 CLIRun 包中，包的类图结构如图 4-17 所示。

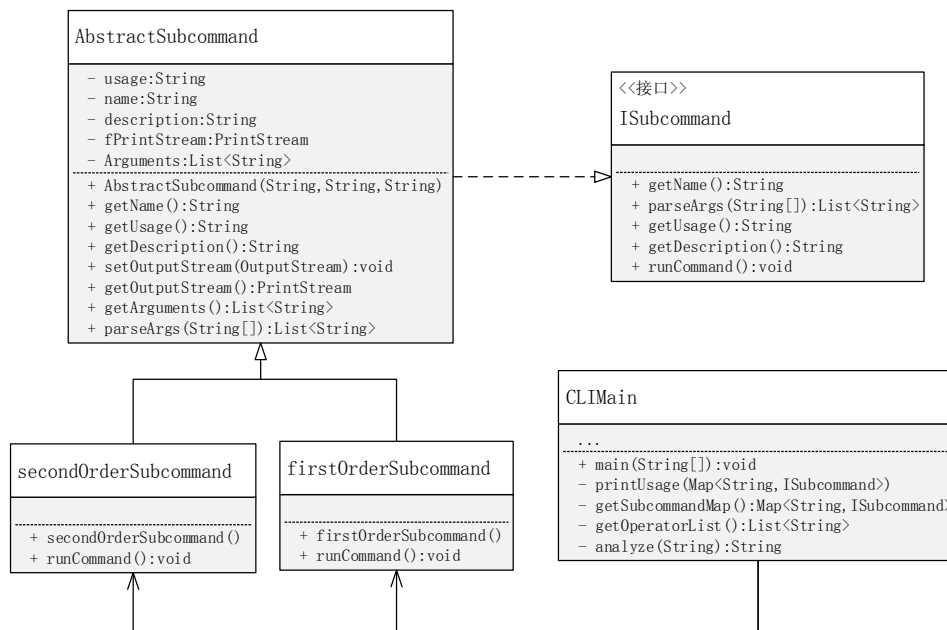


图 4-17 CLIRun 包的类图结构

将 Java Project 导出为可运行 JAR 文件，找到导出的 jar 文件目录，在 META-INF\目录下，打开 MINIFEST.MF 文件，设置主程序入口类名 Main-Class 为 CLIRun.CLIMain，即可在 cmd 中通过[java -jar xx.jar]命令打开导出的 jar 文件，运行系统。使用该命令打开 jar 文件的结果如图 4-18 所示。

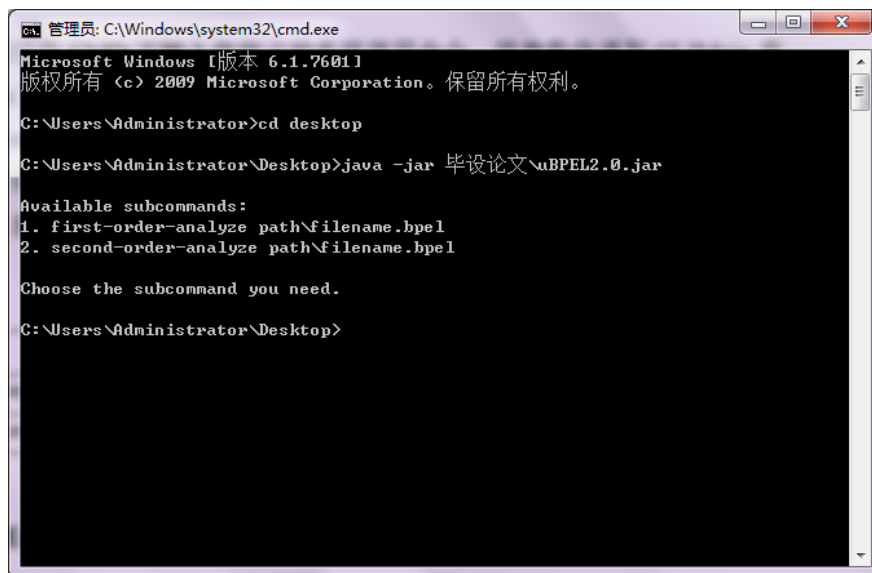


图 4-18 [java -jar xx.jar]命令打开 jar 包

通过在 CMD 下输入自定义的系统使用命令，将输入的命令作为参数传递到 CLIMain 类的 main 方法中，调用 CLIMain 类的 analyze 方法解析传入的命令，通过 generate 包中 GenerateFacade 类提供的统一接口，调用一阶或二

阶变异体生成子系统，完成一阶或二阶变异体的生成。本系统自定义的使用命令如表 4-1 所示。

表 4-1 自定义的系统使用命令

| 名称 | 用法 | 描述 |
|----------------------|--------------------|---|
| first-order-analyze | path\filename.bpel | Generate first-order mutants for your BPEL file. |
| second-order-analyze | path\filename.bpel | Generate second-order mutants for your BPEL file. |

4.2.4 Eclipse 插件实现

Eclipse 平台提供插件开发环境 PDE (Plug-in Development)^[21]，开发人员可根据自己需求开发相应插件，集成到 Eclipse 平台使用。Eclipse 的核心思想^[21]是一切皆为插件，平台提供的大部分功能，均是通过插件实现的。

系统 Eclipse 插件形式开发流程如下：

1) 创建插件项目。

选择“File” — “New” — “Other” — “Plug-in Project”，设置项目名称，选择运行环境，创建一个插件项目 uBPEL。其中，META-INF 目录下的 MANIFEST.MF 文件记录了插件的状态信息，包括插件的依赖关系、运行时的类加载路径以及插件的名称等。plugin.xml 文件通过 XML 文件格式描述扩展点的具体信息，包括扩展点的实现和定义。

2) 创建菜单项。

打开 MANIFEST.MF 文件，在打开的编辑器窗口中选择“Extensions”项，添加扩展点“org.eclipse.ui.actionSets”，并新建操作集“Sample Action Set”，添加菜单项“uBPEL”，并定义菜单项下的操作“Open”。配置结果如图 4-19 所示。配置体现在 plugin.xml 文件中，结果如图 4-20 所示。

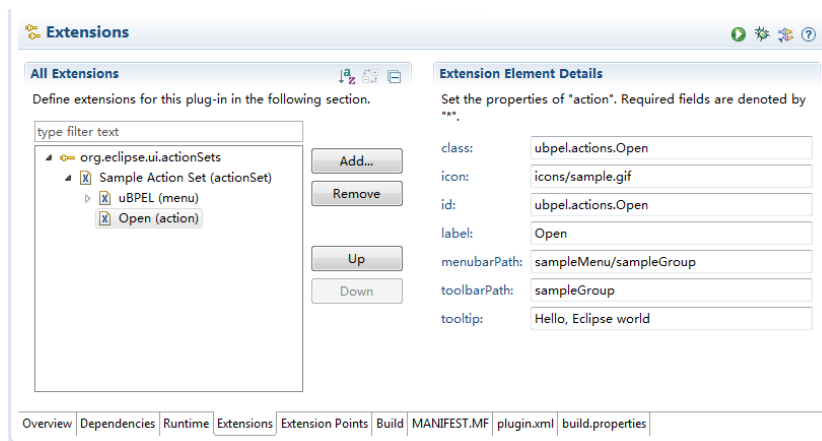


图 4-19 菜单项配置结果

```
<plugin>
  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Sample Action Set"
      visible="true"
      id="uBPPEL.actionSet">
      <menu
        label="uBPPEL"
        id="sampleMenu">
        <separator
          name="sampleGroup">
        </separator>
      </menu>
      <action
        label="Open"
        icon="icons/sample.gif"
        class="ubpel.actions.Open"
        tooltip="Hello, Eclipse world"
        menubarPath="sampleMenu/sampleGroup"
        toolbarPath="sampleGroup"
        id="ubpel.actions.Open">
      </action>
    </actionSet>
  </extension>
</plugin>
```

图 4-20 plugin.xml 文件内容

3) 配置第三方 jar 包。

插件开发环境引用类路径的设置和运行平台引用类路径的设置不同，若不配置第三方 jar 包，会导致运行时出现 `NoClassDefFoundError` 异常。将项目添加到该插件项目中后，选择编辑器窗口的“Runtime”项，把第三方 jar 包添加到“Classpath”面板中，如图 4-21 所示。

4) 构建插件项目。

选择编辑器窗口的“Build”项，在“Binary Build”面板中，选择需要构建的内容，如图 4-22 所示。生成的 `build.properties` 文件内容如图 4-23 所示。

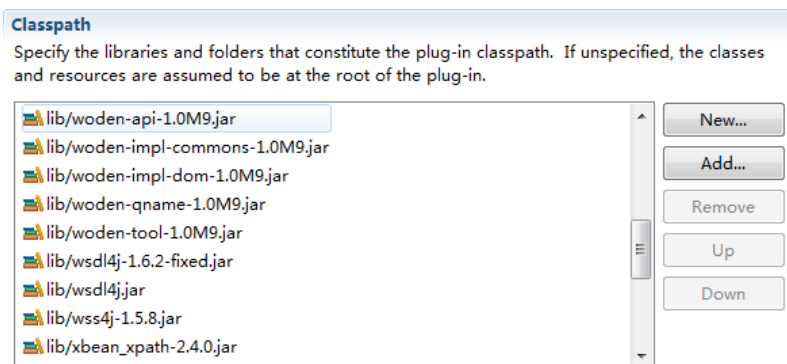


图 4-21 配置第三方 jar 包到 Classpath 面板

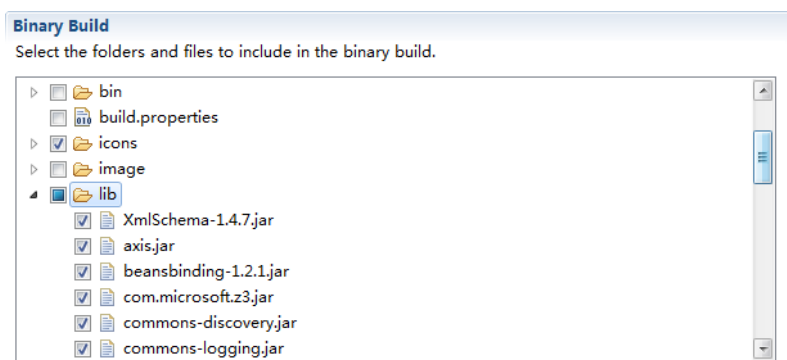


图 4-22 构建插件项目

5) 打包插件。

右键插件项目，选择“Export”—“Deployable plug-ins and fragments”，选择需要打包的项目，设置路径，插件项目即可打包成功。将插件项目打包后的 jar 包拷贝到 Eclipse 的 Plugins 目录下面，重启 Eclipse，即可使用该插件。

```
source.. = src/
output.. = bin/
bin.includes = plugin.xml,\
               META-INF/, \
               .,\
               icons/, \
               lib/axis.jar, \
               lib/beansbinding-1.2.1.jar, \
               lib/com.microsoft.z3.jar, \
               lib/commons-discovery.jar, \
               lib/commons-logging.jar, \
               lib/dom4j-1.6.1.jar, \
               lib/jaxen-1.1-beta-8-no-dom.jar, \
               lib/jaxrpc.jar, \
               lib/log4j-1.2.14.jar, \
               lib/soap-xmlbeans-1.2.jar, \
               lib/soapui-4.0.1.jar, \
               lib/soapui-xmlbeans-4.0.1.jar, \
               lib/woden-ant-1.0M9.jar, \
               lib/woden-api-1.0M9.jar, \
               lib/woden-impl-commons-1.0M9.jar, \
               lib/woden-impl-dom-1.0M9.jar, \
               lib/woden-qname-1.0M9.jar, \
               lib/woden-tool-1.0M9.jar, \
               lib/wsd14j-1.6.2-fixed.jar, \
```

图 4-23 build.properties 文件内容

4.3 工具演示

本节使用 SupplyChain.bpel 程序和 AEL 变异算子作为示例，演示 μ BPEL2.0 变异体生成系统。

- 使用图形用户界面交互形式运行系统

在 Eclipse 平台下运行程序，进入系统初始界面，点击 Start 按钮进入系统功能界面。系统支持对 BPEL 程序的一阶变异体和二阶变异体自动化生成过程。首先进行一阶变异体生成演示。通过点击 First-Order Mutants Generate 按钮，进入一阶变异体生成界面。选择待测的 SupplyChain.bpel 程序和 AEL 变异算子。点击 Finish 按钮。如图 4-24 所示。

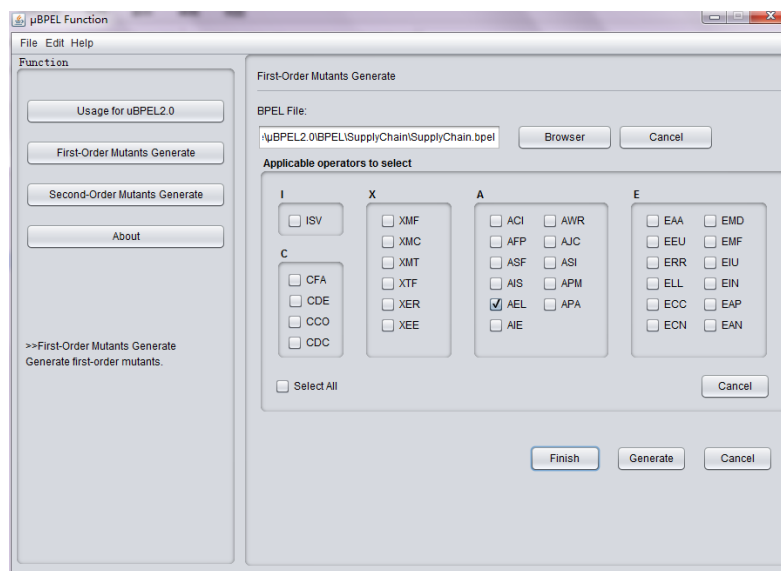


图 4-24 SupplyChain.bpel 的一阶变异体生成

点击 Generate 按钮，此时 BPEL 文件的路径及选择的变异算子作为参数，传递到 run 包的 main 类 test 方法中。借用 GenerateFacade 类接口，调用一阶变异体生成子系统。生成的变异体信息如图 4-25 所示。图中显示生成 AEL 变异体 7 个，SupplyChain.bpel 的文件内容及变异体文件的内容显示在右侧区域。用户可直观的观察变异体文件和原始 BPEL 实例文件的不同。

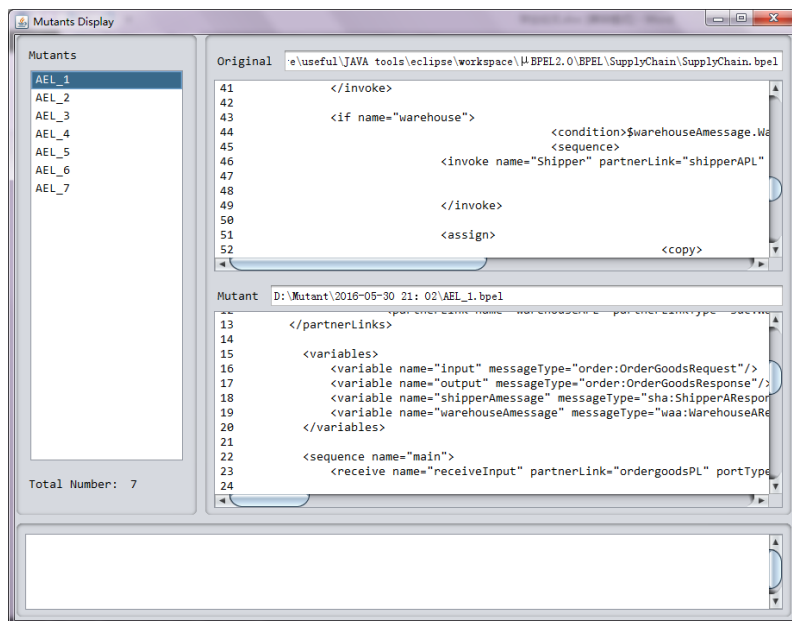


图 4-25 SupplyChain.bpel 生成的一阶变异体

接下来演示二阶变异体的生成, 通过点击 **Second-Order Mutants Generate** 按钮, 进入二阶变异体生成界面。首先选择 **SupplyChain.bpel** 文件和所要应用的 AEL 变异算子, 并选择 **LastToFirst** 组合算法。该算法按照一阶变异体生成顺序, 将一阶变异体首尾依次组合生成二阶变异体。点击 **Finish** 按钮, 完成过程的配置。如图 4-26 所示。

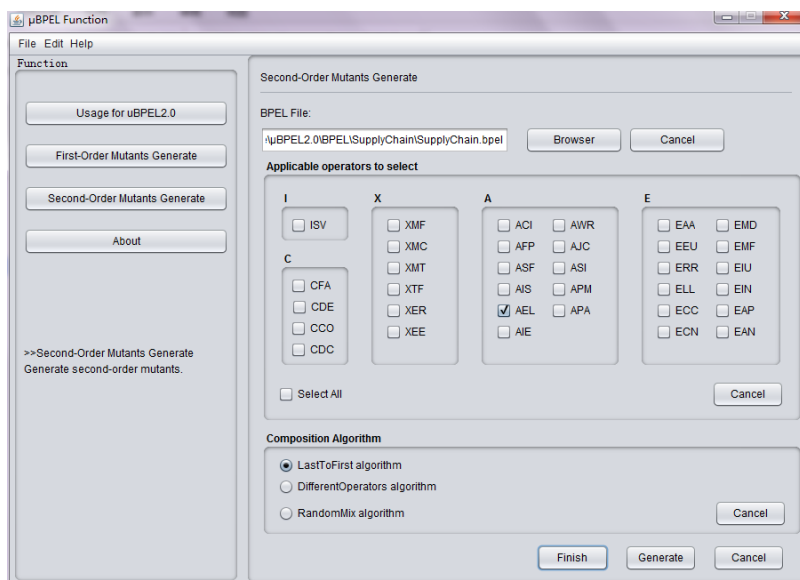


图 4-26 SupplyChain.bpel 的二阶变异体生成

点击 **Generate** 按钮, 系统将获取 BPEL 文件路径, 变异算子及组合算法 **Index** 这些参数, 完成二阶变异体生成。生成的二阶变异体如图 4-27 所示。

二阶变异体以组合的两个一阶变异体的名称进行命名。图中显示生成二阶变异体 4 个，包括 AEL1_AEL7、AEL2_AEL6、AEL3_AEL5、AEL4_AEL5。

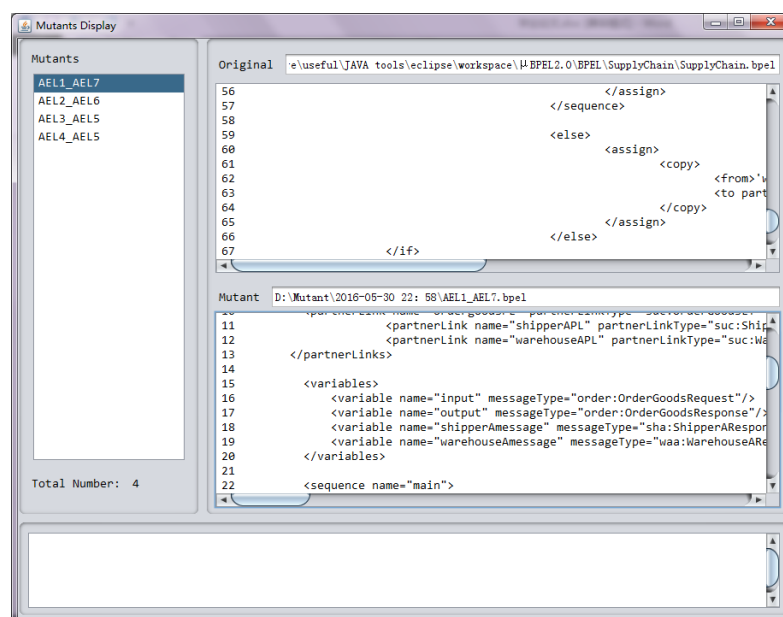


图 4-27 SupplyChain.bpel 生成的二阶变异体

- 命令行交互运行系统

导出可运行的 JAR 文件 uBPEL2.0.jar，使用命令[java -jar uBPEL2.0.jar]打开 jar 包，获得系统的命令使用说明。如图 4-28 所示，该系统提供[first-order-analyze]、[second-order-analyze]两种命令，分别用于生成一阶和二阶变异体。

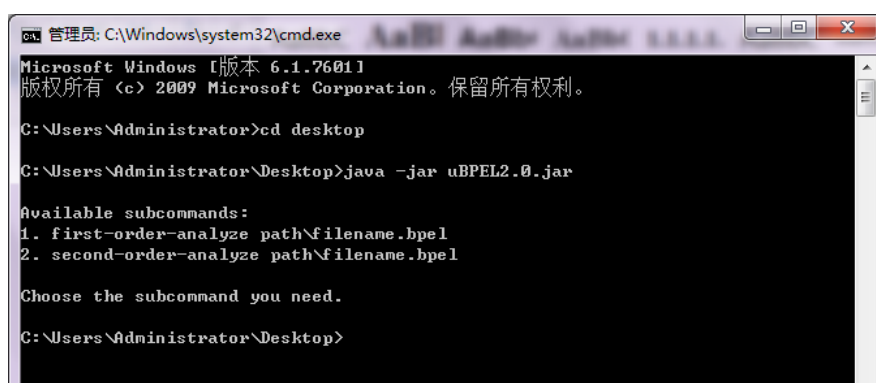
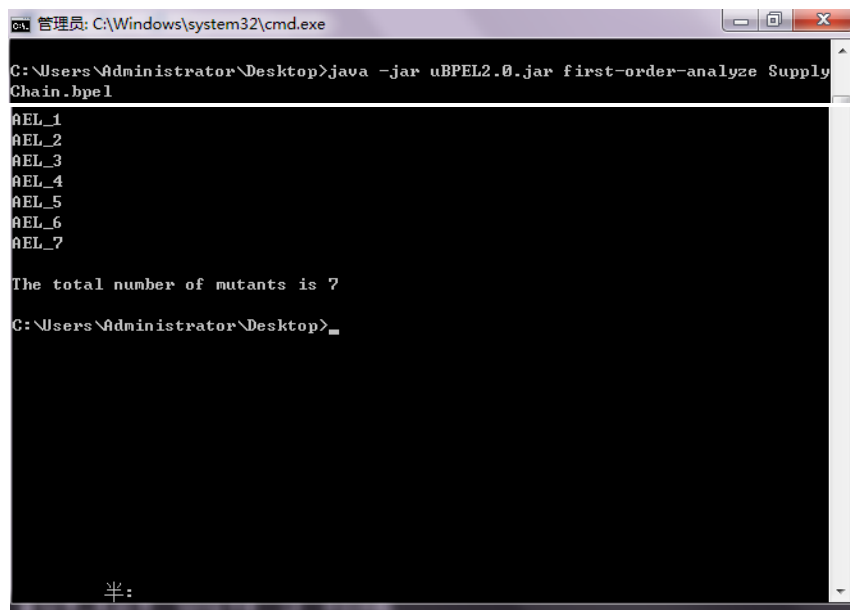


图 4-28 打开 uBPEL2.0.jar 包

通过输入一阶变异体生成命令[first-order-analyze SupplyChain.bpel]，完成一阶变异体的生成。该实例生成 7 个 AEL 类变异体，如图 4-29 所示。

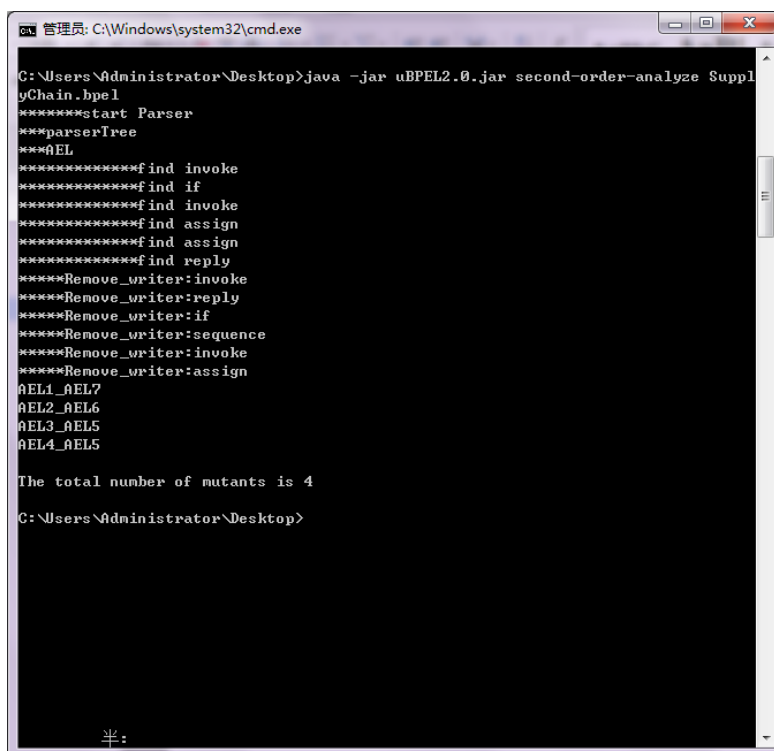


```
管理员: C:\Windows\system32\cmd.exe
C:\Users\Administrator\Desktop>java -jar uBPEL2.0.jar first-order-analyze SupplyChain.bpel
AEL_1
AEL_2
AEL_3
AEL_4
AEL_5
AEL_6
AEL_7

The total number of mutants is 7
C:\Users\Administrator\Desktop>
```

图 4-29 一阶变异体生成

通过输入二阶变异体生成命令[second-order-analyze SupplyChain.bpel],完成二阶变异体的生成。该实例生成 4 个二阶变异体,如图 4-30 所示。



```
管理员: C:\Windows\system32\cmd.exe
C:\Users\Administrator\Desktop>java -jar uBPEL2.0.jar second-order-analyze SupplyChain.bpel
*****start Parser
*****parserTree
*****AEL
*****find invoke
*****find if
*****find invoke
*****find assign
*****find assign
*****find reply
*****Remove_writer:invoke
*****Remove_writer:reply
*****Remove_writer:if
*****Remove_writer:sequence
*****Remove_writer:invoke
*****Remove_writer:assign
AEL1_AEL7
AEL2_AEL6
AEL3_AEL5
AEL4_AEL5

The total number of mutants is 4
C:\Users\Administrator\Desktop>
```

图 4-30 二阶变异体生成

- Eclipse 插件形式集成到 Eclipse 开发平台
- 将导出的插件项目的 jar 包拷贝到 Eclipse 的 Plugins 目录下,重启 Eclipse,

完成插件的安装。在 Eclipse 的菜单栏找到该插件。点击“uBPEL”菜单，选择“Open”菜单项，即可打开 uBPEL2.0 变异体生成系统，如图 4-31 所示。之后系统的操作和图形用户界面交互相同。

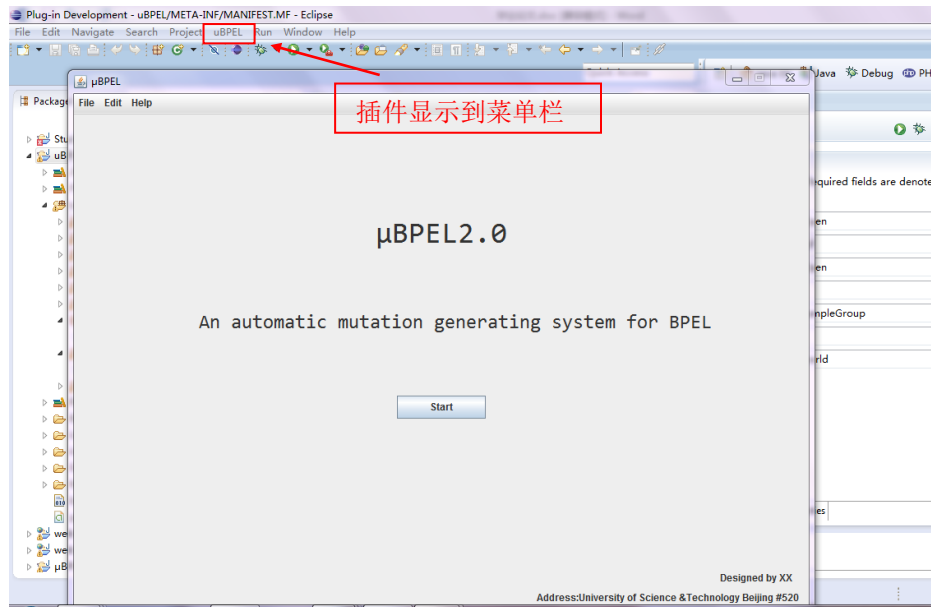


图 4-31 插件项目集成到 Eclipse 开发平台

4.4 小结

本章主要介绍了系统的实现，包括系统功能实现、系统界面实现、命令行交互实现及 Eclipse 插件实现，最终设计完成了一个改进的功能更加多样的面向 BPEL 程序的变异体生成系统，并采用一个 BPEL 程序实例演示了系统的使用，展示了系统各个功能的使用方式。

5 实验评估

本章以六个 BPEL 程序实例测试评估改进的面向 BPEL 程序的变异体生成系统，并与相似工具进行功能比较。

5.1 实验目的

本次实验围绕以下问题展开：

- 1) 验证改进的面向 BPEL 程序的变异体生成系统 μ BPEL2.0 的有效性。
- 2) 与 Estero-Botaro 等人开发的 MuBPEL^[8]系统进行比较，评估本文实现的系统生成变异体集合的完备性。
- 3) 从系统代码的可复用性、系统结构及可扩展性、变异体集合完备性的角度与课题组前期开发的系统进行对比。

5.2 实验设计与过程

5.2.1 实验对象

采用六个 BPEL 程序实例进行验证。分别为：

SupplyChain 实例、SmartShelf 实例、LoanApproval 实例、SupplyCustomer 实例、TravelAgency 实例、CarEstimate 实例。具体如下：

1) SupplyChain 实例

具有 3 个 Web 服务的供应链管理系统。客户输入商品的名称和数量，零售商根据订货单及供应商库存情况向测试人员反馈信息，其流程图如图 5-1 所示。

2) SmartShelf 实例

具有 14 个 Web 服务的货架物品管理系统。输入商品的名称和数量，管理系统查询商品信息后，给测试人员反馈相关信息。其流程图如图 5-2 所示。

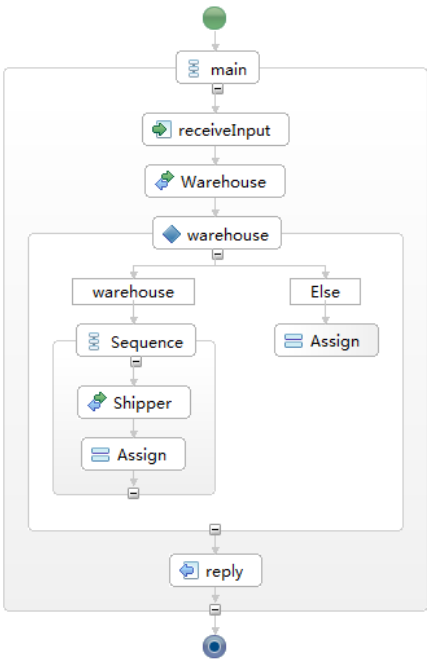


图 5-1 SupplyChain 实例流程

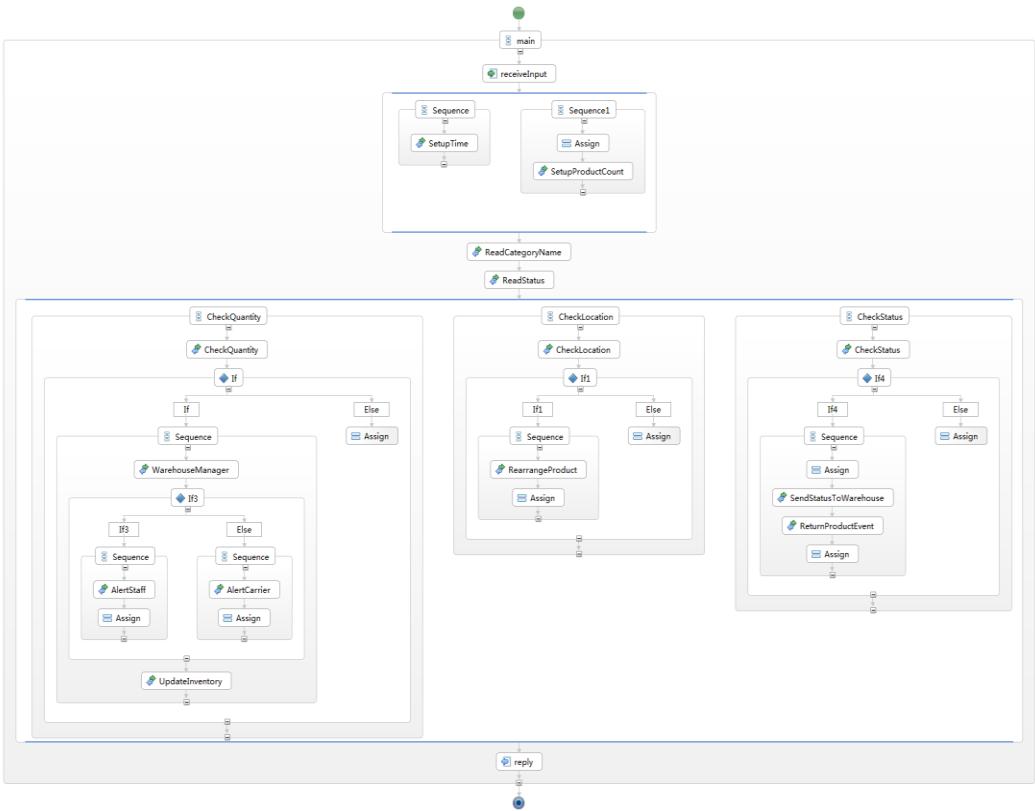


图 5-2 SmartShelf 实例流程

3) LoanApproval 实例

具有 6 个 Web 服务的贷款审批系统。客户输入自己的个人信息和贷款数

目，贷款审批执行审批流程后，返回贷款成功或失败。其流程图如图 5-3 所示。

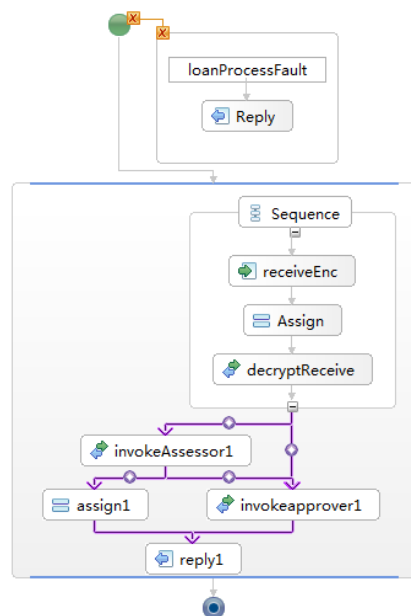


图 5-3 LoanApproval 实例流程

4) SupplyCustomer 实例

具有 6 个 Web 服务的项目订单管理系统。客户输入订单的信息和地址，系统通过验证之后向客户反馈信息，其流程图如图 5-4 所示。

5) TravelAgency 实例

具有 9 个 Web 服务的旅行社预定系统。客户提供个人信息和人数，系统根据人数来选择预订服务，最后将预订信息与结算账单返回给客户。其流程图如图 5-5 所示。

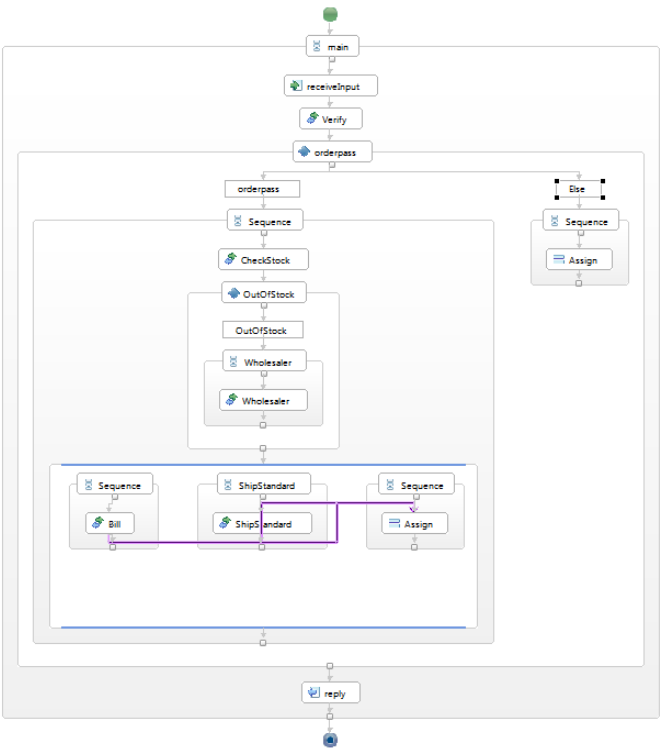


图 5-4 SupplyCustomer 实例流程

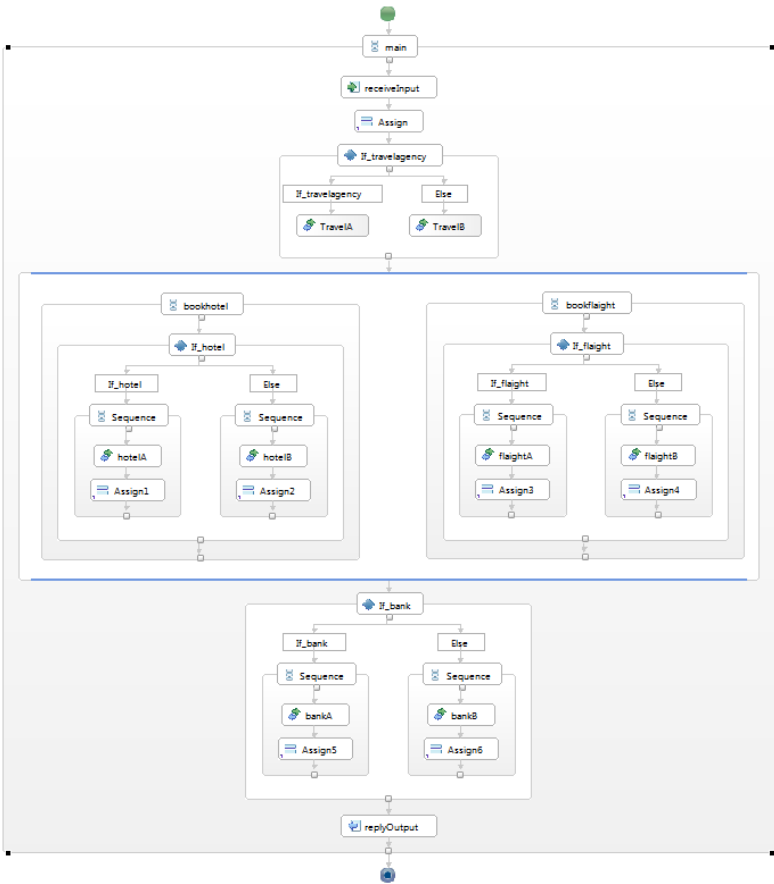


图 5-5 TravelAgency 实例流程

6) CarEstimate 实例

具有 8 个 Web 服务的汽车修复评估系统。顾客提供评估请求，系统首先进行初步评估，然后根据请求对汽车进行简单评估或复杂评估，最后调用最终评估将评估结果返回给顾客。其流程图如图 5-6 所示。

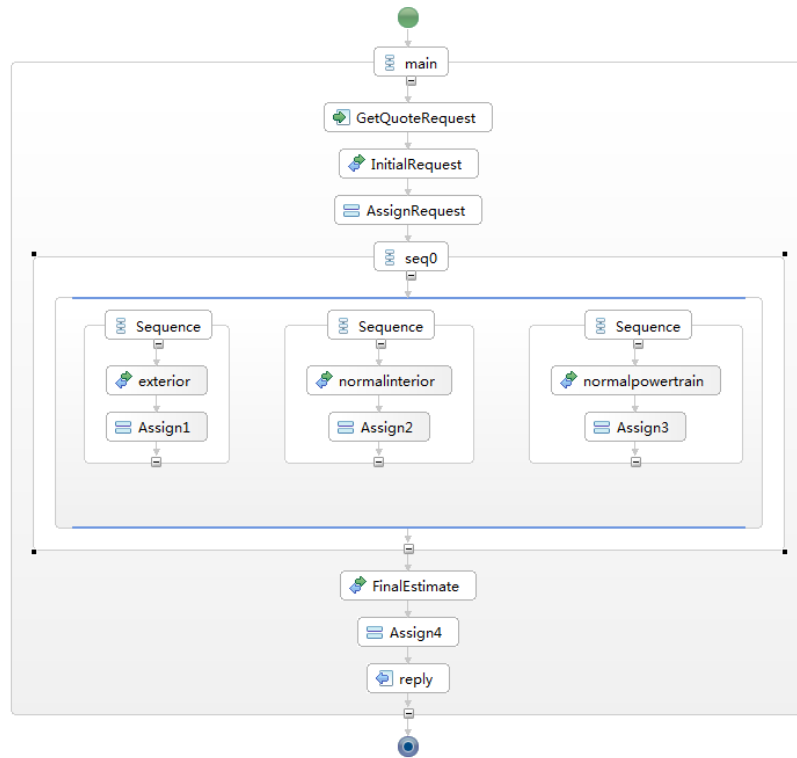


图 5-6 CarEstimate 实例流程

5.2.2 实验过程

实验采用图形用户界面交互的方式，分两部分进行：一阶变异体生成验证和二阶变异体生成验证。步骤分别如下：

1) 一阶变异体生成验证

- ① 针对采用的六个 BPEL 程序实例，分别使用本文改进的系统 μ BPEL2.0 和 MuBPEL 系统生成一阶变异体，统计两个系统生成的一阶变异体的类别及数量。
- ② 根据各变异算子转换规则，评估该系统生成的一阶变异体的正确性。
- ③ 对比该系统与 MuBPEL 系统生成的一阶变异体的类别和数量，评估该系统生成一阶变异体集合的完备性。

2) 二阶变异体生成验证

- ① 针对采用的六个 BPEL 程序实例，使用 μ BPEL2.0 系统生成二阶变异

体,统计系统生成的二阶变异体的类别及数量。

- ② 根据各变异算子转换规则,判断该系统生成的二阶变异体的正确性。

5.3 结果分析与比较

5.3.1 一阶变异体生成验证

针对采用的六个 BPEL 程序实例,该系统与 MuBPEL 系统生成的一阶变异体类别及数量统计如表 5-1 所示。(Mu 代表 MuBPEL 系统,u 代表该系统 uBPEL2.0。)

因生成的一阶变异体较多,以对 SupplyChain.bpel 程序生成的一个 AEL 类的变异体 AEL1.bpel 为例,说明该系统生成的一阶变异体的正确性。AEL 变异算子用于移除 BPEL 程序中的活动。图 5-7 显示了 SupplyChain.bpel 文件与 AEL1.bpel 文件的内容,最下方区域显示了该变异的具体信息,变异位置在 SupplyChain.bpel 文件中高亮显示。

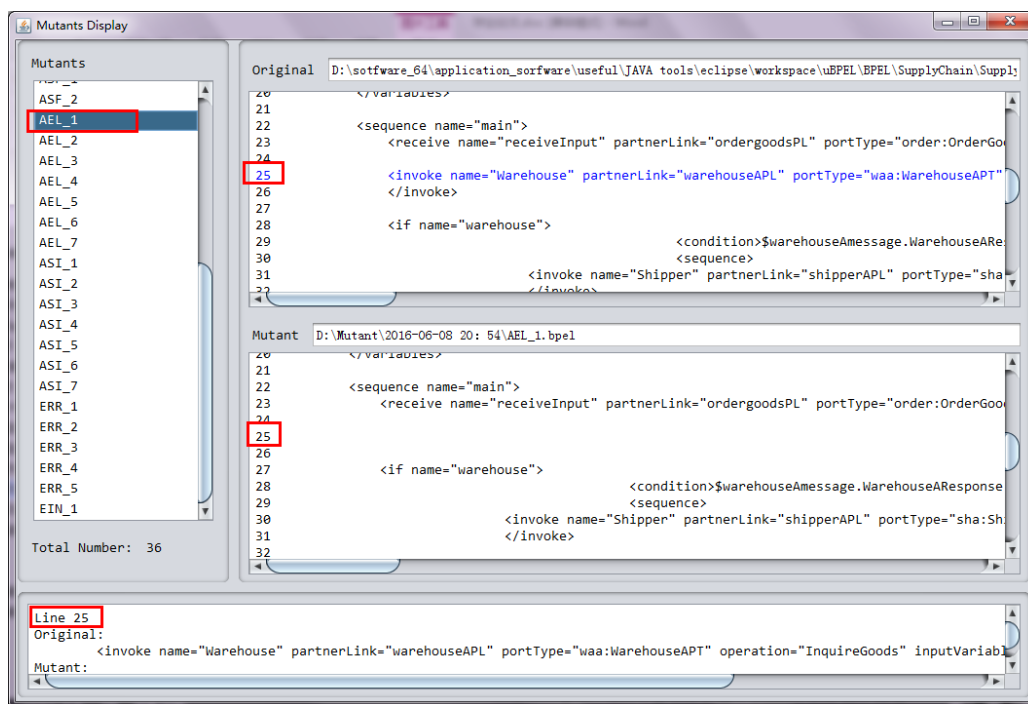


图 5-7 AEL1 变异体信息

由图可以看出, AEL1 变异体移除了 SupplyChain.bpel 程序中第 25 行的 <invoke>活动,变异体生成正确。

表 5-1 uBPEL2.0 系统与 MuBPEL 系统生成的一阶变异体情况统计

| 变 异 算子 | SupplyChain | | SmartShelf | | LoanApproval | | SupplyCustomer | | TravelAgency | | CarEstimate | |
|-----------|-------------|----|------------|-----|--------------|----|----------------|----|--------------|-----|-------------|----|
| | Mu | u | Mu | u | Mu | u | Mu | u | Mu | u | Mu | u |
| ISV | 0 | 0 | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| CFA | 7 | 7 | 40 | 40 | 8 | 9 | 17 | 17 | 30 | 30 | 16 | 16 |
| CDE | 2 | 2 | 8 | 8 | 8 | 8 | 4 | 4 | 8 | 8 | 0 | 0 |
| CCO | 2 | 2 | 8 | 8 | 8 | 8 | 4 | 4 | 8 | 8 | 0 | 0 |
| CDC | 2 | 2 | 8 | 8 | 8 | 8 | 4 | 4 | 8 | 8 | 0 | 0 |
| EAA | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| EEU | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ERR | 5 | 5 | 20 | 20 | 20 | 20 | 10 | 10 | 20 | 20 | 0 | 0 |
| ELL | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ECC | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 4 | 4 | 0 | 0 |
| ECN | 0 | 0 | 0 | 0 | 8 | 8 | 0 | 0 | 16 | 16 | 0 | 0 |
| EMD | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| EMF | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| EIN | 1 | 1 | 4 | 4 | 4 | 4 | 2 | 2 | 4 | 4 | 0 | 0 |
| EIU | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 4 | 0 | 0 | 0 |
| EAP | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 4 | 0 | 0 | 0 |
| EAN | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 4 | 0 | 0 | 0 |
| ACI | --- | 0 | --- | 0 | --- | 0 | --- | 0 | --- | 0 | --- | 0 |
| AFP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ASF | 1 | 2 | 9 | 11 | 0 | 1 | 1 | 6 | 6 | 10 | 3 | 5 |
| AIS | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AEL | 7 | 7 | 40 | 40 | 8 | 9 | 17 | 17 | 30 | 30 | 16 | 16 |
| AIE | 1 | 1 | 4 | 4 | 0 | 0 | 1 | 1 | 4 | 4 | 0 | 0 |
| AWR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AJC | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ASI | 4 | 7 | 26 | 31 | 1 | 3 | 6 | 9 | 16 | 16 | 18 | 18 |
| APM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| APA | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| XMF | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| XMC | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| XMT | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| XTF | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| XER | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| XEE | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 总数 | 32 | 36 | 167 | 174 | 83 | 84 | 66 | 74 | 166 | 158 | 53 | 55 |

对比表 5-1 中 uBPEL2.0 系统与 MuBPEL 系统生成的一阶变异体的类别及数量，我们可以看出，两个系统除个别变异算子生成的变异体数量有差异，其他变异算子生成的变异体数量均相同。

数量存在差异的变异算子为 ISV、CFA、EIU、EAP、EAN、ACI、ASF、

AEL、ASI。结合文献[8]，通过比较两个系统生成的变异体文件的内容，得出 MuBPEL 系统的这些变异算子与 uBPEL2.0 系统生成变异体数量不同的原因。具体原因如表 5-2 所示。

表 5-2 个别变异算子生成变异体数量不同的原因

| 变异算子 | 原因 |
|------|------------------------------------|
| ISV | 部分位置的变量未替换 |
| CFA | <process>下的第一个<flow>未替换 |
| EIU | 在常数值前插入“-” |
| EAP | 将常数值替换为其正值 |
| EAN | 将常数值替换为其负值 |
| ACI | 不支持该变异算子 |
| ASF | <process>下的第一个<sequence>未替换 |
| AEL | <process>下的第一个<flow>和<sequence>未移除 |
| ASI | <sequence>下第一个<receive>不参与交换 |

通过进一步分析这些原因，我们发现 ISV、CFA、ACI、ASF、AEL、ASI 变异算子这些不采取的操作是为了保证生成的变异体文件的安全性和有效性。EIU、EAP、EAN 变异算子的这些操作则是为了保证变异覆盖的全面性。

由以上分析我们可以得出：

- 1) 该系统生成的一阶变异体是正确的，系统一阶变异体生成功能有效。
- 2) 通过对比该系统与 MuBPEL 系统生成的一阶变异体的类别和数量，分析个别变异算子差别存在的原因，我们可以认为该系统生成的一阶变异体集合的完备性较好。

5.3.2 二阶变异体生成验证

针对采用的六个 BPEL 程序实例，使用该系统生成二阶变异体，统计系统生成的二阶变异体的类别及数量。图 5-8 显示的是使用 LastToFirst 组合算法生成的二阶变异体情况。该算法按照一阶变异体的生成顺序，将一阶变异体首尾组合，生成二阶变异体。所以，按此组合算法生成的二阶变异体的数量约为一阶变异体数量的一半。

结合各 BPEL 实例生成的一阶变异体的类别和数量，综合图 5-8 显示的二阶变异体的生成情况，我们可以得出：

该系统生成的二阶变异体是正确的，系统二阶变异体生成功能有效。

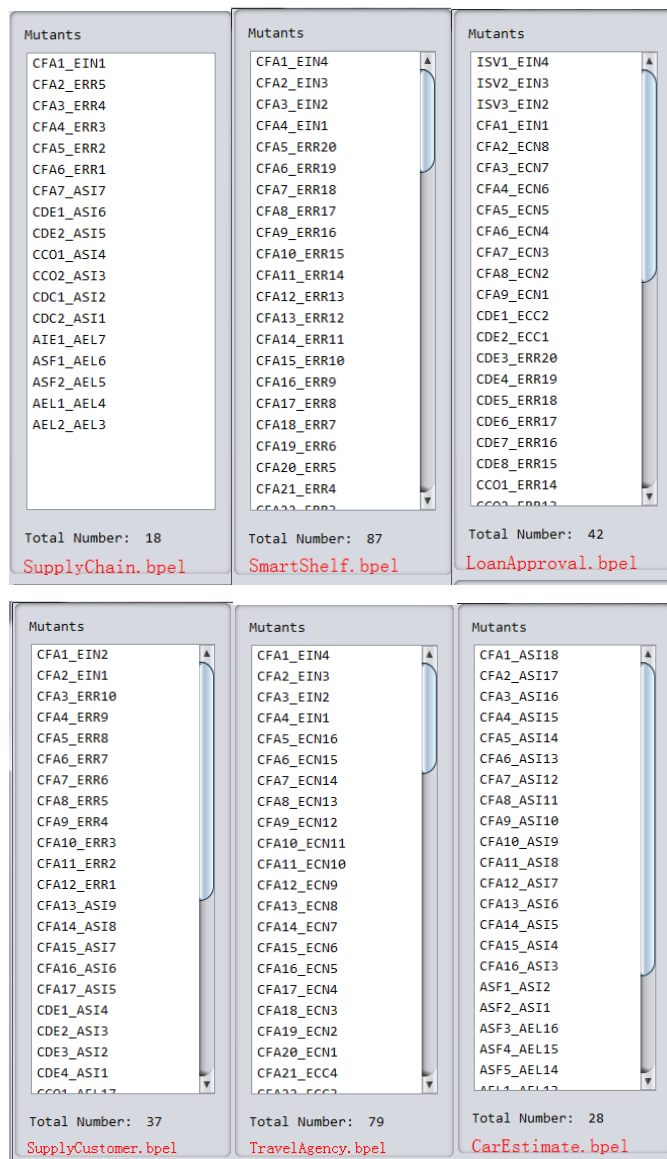


图 5-8 系统生成的二阶变异体类别及数量

5.3.3 与前期开发系统的对比

将本文改进的系统 μ BPEL2.0 与课题组前期研发的面向 BPEL 程序的变异体生成系统^[10]进行对比, 可以得出该系统优于前期开发的系统。具体总结如下:

- 1) 采用访问者模式、组合模式和外观模式对系统进行重构与优化, 改进系统结构, 降低客户端与子系统之间的耦合性, 使得系统的结构更加清晰。
- 2) 前期开发的系统只支持 Estero-Botaro 等人^[3]提出的 26 种变异算子, 改进的系统可支持 34 种变异算子, 而且通过与 MuBPEL 系统生成的一阶变异体进行对比, 可以看出系统变异体集合的完备性较好。

- 3) 通过将转换规则相似的变异算子进行归类和抽象,改进变异算子匹配及生成程序,减少了系统的代码冗余,改善了代码结构,提高了代码的可复用性。
- 4) 前期开发的系统若添加或修改变异算子,扩展高阶变异体生成功能,需对系统进行大幅度修改,系统的扩展性较差。本文通过利用 DOM4J 解析工具的 **Visitor** 类及 **XMLWriter** 类,将访问者模式和组合模式应用到变异算子匹配及处理程序中,并扩展添加了二阶变异体生成功能,增强了系统的可扩展性,为日后增加变异算子种类及扩展系统高阶变异体生成功能提供了便利。
- 5) 改进的系统支持图形用户界面交互、命令行交互及 **Eclipse** 插件三种不同方式的集成和使用,更加便于用户使用。

5.4 小结

采用六个 **BPEL** 程序实例对改进后的系统进行一阶变异体和二阶变异体生成验证,并与 **MuBPEL** 系统和课题组前期开发的系统进行比较。通过对实验结果的分析 and 比较,我们可以看出改进的系统生成的变异体的正确性和完备性较好。和课题组前期开发的系统相比,本文改进的系统的系统结构更加清晰,系统的可复用性、可扩展性和可维护性增强。

6 结 论

本文在面向 BPEL 程序的变异测试理论的基础上, 针对课题组研发的面向 BPEL 程序的变异体生成系统存在的问题, 采用设计模式对系统进行重构与优化, 设计并实现一个改进的支持不同方式集成与使用的系统。并采用多个 BPEL 程序实例测试评估系统。

本文的主要成果总结如下:

- 采用设计模式重构和优化了面向 BPEL 程序的变异体生成系统, 改进了变异算子匹配及变异体生成程序。
- 设计实现了一个改进的变异体生成系统, 可支持图形用户界面交互、命令行交互及 Eclipse 插件三种不同方式的集成和使用。
- 采用 BPEL 程序实例验证了系统的变异体生成功能。

当前研究工作存在的不足及未来的工作:

- 该系统生成的变异体中存在不安全或无效的变异体, 需要进一步研究, 过滤掉不安全或者无效的变异体。
- 可以探究生成二阶变异体的组合算法, 研究不同组合算法生成二阶变异体的数量及有效性。
- 进一步优化系统的界面及插件功能, 提高工具的界面友好性。

参考文献

- [1] Juric M B, Mathew B, Sarang P G Business Process Execution Language for Web Services: An Architect and Developer's Guide to Orchestrating Web Services Using BPEL4WS [M]. Packt Publishing Ltd, 2006.
- [2] DeMillo R A, Lipton R J, Sayward F G Hints on Test Data Selection: Help for the Practicing Programmer [J]. IEEE Computer, 1978, 11(4):34-41.
- [3] Estero-Botaro A, Palomo-Lozano F, Medina-Bulo I. Mutation Operators for WS-BPEL 2.0 [C]. Proceedings of the 21th International Conference on Software & Systems Engineering and their Applications, 2008:1-7.
- [4] 胡荣, 王巧玲, 孙昌爱, 等. MuBPEL:一个面向 BPEL 的变异体自动生成系统 [EB/OL]. 北京: 中国科技论文在线. <http://www.paper.edu.cn/releasepaper/content/201411-445>.
- [5] King K N, Offutt A J. A fortran language system for mutation-based software testing [J]. Software: Practice and Experience, 1991, 21(7): 685-718.
- [6] Ma Y S, Kwon Y R, Offutt J. Inter-class mutation operators for Java [C]. Proceedings of the 13th International Symposium on Software Reliability Engineering, 2002: 352-363.
- [7] Ma Y S, Offutt J, Kwon Y R. MuJava: an automated class mutation system [J]. Software Testing, Verification and Reliability, 2005, 15(2): 97-133.
- [8] Ma Y S, Offutt J, Kwon Y R. MuJava: a mutation system for Java [C]. Proceedings of the 28th international conference on Software engineering. ACM, 2006: 827-830.
- [9] Estero-Botaro A, Palomo-Lozano F, Medina-Bulo I, et al. Quality metrics for mutation testing with applications to WS-BPEL compositions [J]. Software Testing, Verification and Reliability, 2015, 25(5-7): 536-571.
- [10] OASIS. Web Services Business Process Execution Language Version 2.0 [EB/OL]. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, 2007.
- [11] 姚世军, 谢蕾, 吴之铁. 基于 WS-BPEL 的服务组合流程建模工具的设计与实现 [J]. 计算机工程与设计, 2009, 30(17): 3999-4001.
- [12] Hamlet R G Testing programs with the aid of a compiler [J]. IEEE Transactions on Software Engineering, 1977, 3(4): 279-290.
- [13] Jia Y, Harman M. An analysis and survey of the development of mutation

- testing [J]. IEEE Transactions on Software Engineering, 2011, 37(5): 649-678.
- [14] 陈翔, 顾庆. 变异测试: 原理、优化和应用 [J]. 计算机科学与探索, 2012, 6(12): 1057-1075.
- [15] 肖袁. 基于 DOM4J 的 XML 文档解析技术[J]. 科技信息, 2011, (2): 229-230.
- [16] XML DOM 教程 [EB/OL]. <http://www.w3school.com.cn/xml/dom/index.asp>.
- [17] Vlissides J, Helm R, Johnson R, Vlissides J. Design patterns: Elements of reusable object-oriented software [M]. Reading: Addison-Wesley, 1995.
- [18] 刘径舟, 张玉华, 等. 设计模式其实很简单 [M]. 北京: 清华大学出版, 2013.
- [19] 吴亚峰, 纪超. Java SE 6.0 编程指南 [M]. 人民邮电出版社, 2007.
- [20] Java 反射机制 [EB/OL]. <http://blog.csdn.net/kaoa000/article/details/8453371>, 2013.
- [21] 罗强. 基于 Eclipse 平台的插件开发 [J]. 计算机光盘软件与应用, 2012 (4): 153-154.
- [22] Gopinath R, Alipour M A, Ahmed I, et al. On the limits of mutation reduction strategies [C]. Proceedings of the 38th international conference on software engineering. ACM, 2016: 511-522.
- [23] Sun C, Shang Y, Zhao Y, et al. Scenario-Oriented Testing for Web Service Compositions Using BPEL [C]. Proceedings of 12th International Conference on Quality Software (QSIC 2012), 2012: 171-174.
- [24] dom4j 2.0 [EB/OL]. <http://dom4j.sourceforge.net/>, 2010.
- [25] 范春梅, 王新刚, 张卫华. XML 基础教程 [M]. 北京: 人民邮电出版社, 2009.
- [26] Boubeta-Puig J, Medina-Bulo I, García-Domínguez A. Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages [C]. Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011: 398-407.
- [27] Domínguez-Jiménez J J, Estero-Botaro A, García-Domínguez A, et al. GAmara: an automatic mutant generation system for WS-BPEL compositions [C]. Proceedings of the 7th IEEE European Conference on Web Services, 2009: 97-106.

附录A 外文文献原文

On The Limits of Mutation Reduction Strategies

Rahul Gopinath Oregon State University gopinath@eecs.orst.edu

Mohammad Amin Alipour Oregon State University alipour@eecs.orst.edu

Iftekhar Ahmed Oregon State University ahmedi@onid.orst.edu

Carlos Jensen Oregon State University cjensen@eecs.orst.edu

Alex Groce Oregon State University agroce@gmail.com

● ABSTRACT

Although mutation analysis is considered the best way to evaluate the effectiveness of a test suite, hefty computational cost often limits its use. To address this problem, various mutation reduction strategies have been proposed, all seeking to reduce the number of mutants while maintaining the representativeness of an exhaustive mutation analysis. While research has focused on the reduction achieved, the effectiveness of these strategies in selecting representative mutants, and the limits in doing so have not been investigated, either theoretically or empirically.

We investigate the practical limits to the effectiveness of mutation reduction strategies, and provide a simple theoretical framework for thinking about the absolute limits. Our results show that the limit in improvement of effectiveness over random sampling for real-world open source programs is a mean of only 13.078%. Interestingly, there is no limit to the improvement that can be made by addition of new mutation operators.

Given that this is the maximum that can be achieved with perfect advance knowledge of mutation kills, what can be practically achieved may be much worse. We conclude that more effort should be focused on enhancing mutations than removing operators in the name of selective mutation for questionable benefit.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging
Testing Tools

General Terms: Measurement, Verification

Keywords: software testing, statistical analysis, theoretical analysis, mutation analysis

● 1 INTRODUCTION

The quality of software is a pressing concern for the software industry, and is usually determined by comprehensive testing. However, tests are themselves programs, (usually) written by human beings, and their quality needs to be monitored to ensure that they in fact are useful in ensuring software quality (e.g., it is important to determine if the tests are also a quality software system).

Mutation analysis [11, 36] is currently the recommended method [5] for evaluating the efficacy of a test suite. It involves systematic transformation of a program through the introduction of small syntactical changes, each of which is evaluated against the given test

suite. A mutant that can be distinguished from the original program by the test suite is deemed to have been killed by the test suite, and the ratio of all such mutants to the set of mutants identified by a test suite is its mutation (kill) score, taken as an effectiveness measure of the test suite.

Mutation analysis has been validated many times in the past. Andrews et al. [4,5], and more recently Just et al. [32], found that faults generated through mutation analysis resemble real bugs, their ease of detection is similar to that of real faults, and most importantly for us, a test suite's effectiveness against mutants is similar to its effectiveness against real faults.

However, mutation analysis has failed to gain widespread adoption in software engineering practice due to its substantial computational requirements — the number of mutants generated needs to be many times the number of program tokens in order to achieve exhaustive coverage of even first order mutants (involving one syntactic change at a time), and each mutant needs to be evaluated by a potentially full test suite run. A number of strategies have been proposed to deal with the computational cost of mutation analysis. These have been classified [44] orthogonally into do faster, do smarter, and do fewer approaches, corresponding to whether they improve the speed of execution of a single mutant, parallelize the evaluation of mutants, or reduce the number of mutants evaluated.

A large number of do fewer strategies — mutation reduction methods that seek to intelligently choose a smaller, representative, set of mutants to evaluate — have been investigated in the past. They are broadly divided into operator selection strategies, which seek to identify the smallest subset of mutation operators that generate the most useful mutants [43,46], and strata sampling [1,9] techniques, which seek to identify groups of mutants that have high similarity between them to reduce the number of mutants while maintaining representativeness and diversity [53, 54]. Even more complex methods using clustering [19, 37], static analysis [28, 34] and other intelligent techniques [48] are under active research [20].

These efforts raise an important question: What is the actual effectiveness of a perfect mutation reduction strategy over the baseline — random sampling — given any arbitrary program?

We define the efficiency of a selection technique as the amount of reduction achieved, and the effectiveness as the selection technique's ability to choose a representative reduced set of mutants, that require as many test cases to kill as the original set of mutants. The ratio of effectiveness of a technique to that of random sampling is taken as the utility of the technique.

We approach these questions from two directions. First, we consider a simple theoretical framework in which to evaluate the improvement in effectiveness for the best mutation reduction possible, using a few simplifying assumptions, and given oracular knowledge of mutation kills. This helps set the base-line. Second, we empirically evaluate the best mutation reduction possible for a large number of projects, given post hoc (that is, oracular) detection knowledge. This gives us practical (and optimistic) limits given common project characteristics.

Our contributions are as follows:

- We find a theoretical upper limit for the effectiveness of mutation reduction strategies of 58.2% for a uniform distribution of mutants — the distribution most favorable for random

sampling. We later show that for real world programs, the impact of distribution is very small (4.467%) suggesting that uniform distribution is a reasonable approximation.

- We find an empirical upper limit for effectiveness through the evaluation of a large number of open source projects, which suggests a maximum practical utility of 13.078% on average, and for 95% of projects, a maximum utility between 12.218% and 14.26% (one sample u-test $p < 0.001$).

- We show that even if we consider a set of mutants that are distinguished by at least by one test (thus discounting the impact of skew in redundant mutants) we can expect a maximum utility of 17.545% on average, and for 95% of projects, a maximum utility between 16.912% and 18.876% (one sample u-test $p < 0.001$).

What do our results mean for the future of mutation reduction strategies? Any advantage we gain over random sampling is indeed an advantage, however small. However, our understanding of mutant semiotics² is as yet imperfect, and insufficient to infer whether the kind of selection employed is advantageous. In fact, our current research [24] shows that current operator selection strategies seldom provide any advantage over random sampling, and even strata sampling based on program elements never achieves more than a 10% advantage over pure random sampling. Our results suggest that the effort spent towards improving mutant selection mechanisms should be carefully weighed against the potential maximum utility, and the risks associated with actually making things worse through biased sampling.

Our research is also an endorsement of the need for further research into new mutators. It suggests that addition of new mutators and then randomly sampling the same number of mutants as that of the original set, is only subject to a similar maximum disadvantage (upper limit for 95% projects), while having essentially no upper bound on advantage due to increase in effectiveness.

The asymmetry between improvement obtained by operator removal and operator addition is caused by the difference in population from which the random comparison sample is drawn. For operator selection, the perfect set remaining after removal of operators is a subset of the original population. Since the random sample is drawn from the original population, it can potentially contain a mutant from each strata in the perfect set. For operator addition, the new perfect set is a superset of the original population, with as many new strata as there are new mutants (no bounds on the number of new strata). Since the random sample is constructed from the original population, it does not contain the newly added strata. Our results suggest a higher payoff in finding newer categories of mutations, than in trying to reduce the mutation operators already available.

In the interests of easy replication, our research is organized and reproducible using Knitr. The raw Knitr source of our paper along with the R data set required to build the paper, and the instructions to do so, are available [23].

● 2 RELATED WORK

According to Mathur [39], the idea of mutation analysis was first proposed by Richard Lipton, and formalized by DeMillo et al. [17] A practical implementation of mutation analysis

was done by Budd et al. [10] in 1980.

Mutation analysis subsumes different coverage measures [9, 40, 45]; the faults produced are similar to real faults in terms of the errors produced [15] and ease of detection [4, 5]. Just et al. [32] investigated the relation between mutation score and test case effectiveness using 357 real bugs, and found that the mutation score increased with effectiveness for 75% of cases, which was better than the 46% reported for structural coverage.

Performing a mutation analysis is usually costly due to the large number of test runs required for a full analysis [31]. There are several approaches to reducing the cost of mutation analysis, categorized by Offutt and Untch [44] as: do fewer, do smarter, and do faster. The do fewer approaches include selective mutation and mutant sampling, while weak mutation, parallelization of mutation analysis, and space/time trade-offs are grouped under the umbrella of do smarter. Finally, the do faster approaches include mutant schema generation, code patching, and other methods.

The idea of using only a subset of mutants was conceived along with mutation analysis itself. Budd [9] and Acree [1] showed that even 10% sampling approximates the full mutation score with 99% accuracy. This idea was further explored by Mathur [38], Wong et al. [50, 51], and Offutt et al. [43] using Mothra [16] for Fortran.

A number of studies have looked at the relative merits of operator selection and random sampling criteria. Wong et al. [50] compared x% selection of each mutant type with operator selection using just two mutation operators and found that both achieved similar accuracy and reduction (80%). Mresa et al. [41] used the cost of detection as a means of operator selection. They found that if a very high mutation score (close to 100%) is required, x% selective mutation is better than operator selection, and, conversely, for lower scores, operator selection would be better if the cost of detecting mutants is considered.

Zhang et al. [54] compared operator-based mutant selection techniques to random sampling. They found that none of the selection techniques were superior to random sampling. They also found that uniform sampling is more effective for larger programs compared to strata sampling on operators³, and the reverse is true for smaller programs. Recently, Zhang et al. [53] confirmed that sampling as few as 5% of mutants is sufficient for a very high correlation (99%) with the full mutation score, with even fewer mutants having a good potential for retaining high accuracy. They investigated eight sampling strategies on top of operator-based mutant selection and found that sampling strategies based on program components (methods in particular) performed best.

Some studies have tried to find a set of sufficient mutation operators that reduce the cost of mutation but maintain correlation with the full mutation score. Offutt et al. [43] suggested an n-selective approach with step-by-step removal of operators that produce the most numerous mutations. Barbosa et al. [8] provided a set of guidelines for selecting such mutation operators. Namin et al. [42, 47] formulated the problem as a variable reduction problem, and found that just 28 out of 108 operators in Proteum were sufficient for accurate results.

Using only the statement deletion operator was first suggested by Untch [49], who found that it had the highest correlation ($R^2 = 0.97$) with the full mutation score compared to other operator selection methods, while generating the smallest number of mutants. This was

further reinforced by Deng et al. [18] who defined deletion for different language elements, and found that an accuracy of 92% is achieved while reducing the number of mutants by 80%.

A similar mutation reduction strategy is to cluster similar mutations together [20, 27], which has been attempted based on domain analysis [28] and machine learning techniques based on graphs [48].

In operator and mutant subsumption, operators or mutants that do not significantly differ from others are eliminated. Kurtz et al. [35] found that a reduction of up to 24 times can be achieved using subsumption alone, even though the result is based on an investigation of a single program, *cal*. Research into subsumption of mutants also includes Higher Order Mutants (HOM), whereby multiple mutations are introduced into the same set of mutants, reducing the number of individual mutants by subsuming component mutants. HOMs were investigated by Jia et al. [29, 30], who found that they can reduce the number of mutants by 50%.

Ammann et al. [3] observe that the set of minimal mutants corresponding to a minimal test suite has the same cardinality as the test suite, and provides a simple algorithm for finding both a minimal test suite and a corresponding minimal mutant set. Their work also suggests this minimal mutant set as a way to evaluate the quality of a mutation reduction strategy. Finally, Ammann et al. also found that the particular strategies examined are rather poor when it comes to selecting representative mutants. Our work is an extension of Ammann et al. [3] in that we provide a theoretical and empirical bound to the amount of improvement that can be expected by any mutation reduction strategy.

In comparison with previous work [53, 54] our analysis is backed by theory and compares random sampling to the limit of selection. That is, the results from our study are applicable to techniques such as clustering using static analysis, and even improved strata sampling techniques. Further, we are the first to evaluate the effectiveness of non-adequate test suites (Zhang et al. [53] evaluates only the predictive power of non-adequate test suites, not effectiveness). Finally, previous research [53,54] does not compare the effectiveness of the same number of mutants for sampling and operator selection, but rather different operator-selections with samples of increasing size such as 5%, 10% etc. We believe that practitioners will be more interested in comparing the effectiveness achieved by the same numbers of mutants.

● 3 THEORETICAL ANALYSIS

The ideal outcome for a mutation reduction strategy is to find the minimum set of mutants that can represent the complete set of mutants. A mutation reduction strategy accomplishes this by identifying redundant mutants and grouping them together so that a single mutant is sufficient to represent the entire group. The advantage of such a strategy over random sampling depends on two characteristics of the mutant population. First, it depends on the number of redundant mutants in each group of such mutants. Random sampling works best when these groups have equal numbers of mutants in them (uniform distribution), while any other distribution of mutants (skew) results in lower effectiveness of random sampling. However, this distribution is dependent on the program being evaluated. Since our goal is to find the mean advantage for a perfect strategy for an arbitrary program, we use the conservative distribution (uniform) of mutants for our theoretical analysis (we show later that the actual impact of this skew is less than 5% for real world mutants).

The next consideration regards the minimum number of mutants required to represent the entire population of mutants. If a mutant can be distinguished from another in terms of tests that detect it, then we consider both to be distinguishable from each other in terms of faults they represent, and we pick a representative from each set of indistinguishable mutants. Note that, in the real world, the population of distinguishable mutants is often larger than the minimum number of mutants required to select a minimum test suite⁴ able to kill the entire mutant population. This is because while some mutants are distinguishable from others in terms of tests that detect them, there may not be any test that uniquely kills them⁵. Since this is external to the mutant population, and also because such a minimum set of mutants does not represent the original population fully (we can get away with a lower number only because the test suite is inadequate), we assume that distinguishable mutants are uniquely identified by test cases. We note however, that having inadequate test suites favors random sampling, and hence lowers the advantage for a perfect mutation reduction strategy, because random sampling can now miss the mutant without penalty. We derive the limits of mutation reduction for this system using the best strategy possible, given oracular knowledge of mutant kills.

Impact of deviations of parameters:

Skew: The presence of skew reduces the effectiveness of random sampling, and hence increases the utility of the perfect strategy.

Distinguishability: Any distinguishable mutant that is not chosen by the strategy (due to not having a unique detecting test case) decreases the effectiveness of the selection strategy, decreasing its utility.

Before establishing a theoretical framework for utility of mutation reduction strategies, we must establish some terminology for the original and reduced mutant sets and their related test suites.

Terminology: Let M and M_{strategy} denote the original set of mutants and the reduced set of mutants, respectively. The mutants from M killed by a test suite T are given by $\text{kill}(T, M)$ (We use M_{killed} as an alias for $\text{kill}(T, M)$). Similarly the tests in T that kill mutants in M are given by $\text{cover}(T, M)$.

$$\text{kill} : T \times M \rightarrow M$$

$$\text{cover} : T \times M \rightarrow T$$

The test suite T_{strategy} can kill all mutants in M_{strategy} . That is, $\text{kill}(T_{\text{strategy}}, M_{\text{strategy}}) = M_{\text{strategy}}$. If it is minimized with respect to the mutants of the strategy, we denote it by $T_{\text{strategy}}^{\min}$.

Two mutants m and m' are distinguished if the tests that kill them are different: $\text{cover}(T, \{m\}) \neq \text{cover}(T, \{m'\})$.

We use $M_{\text{killed}}^{\text{uniq}}$ to denote the set of distinguished mutants from the original set such that $\forall m, m' \in M \text{ cover}(T, \{m\}) \neq \text{cover}(T, \{m'\})$.

The utility (U_{strategy}) of a strategy is improvement in effectiveness due to using that strategy compared to the base-line (the baseline is random sampling of the same number⁶ of mutants). That is,

$$U_{\text{strategy}} = \left| \frac{\text{kill}(T_{\text{strategy}}^{\min}, M)}{\text{kill}(T_{\text{random}}^{\min}, M)} \right| - 1$$

Note that $T_{\text{strategy}}^{\min}$ is minimized over the mutants selected by the strategy, and it is then applied against the full set of mutants (M) in $\text{kill}(T_{\text{strategy}}^{\min}, M)$.

This follows the traditional evaluation of effectiveness, which goes as follows: start with the original set of mutants, and choose a subset of mutants according to the strategy. Then select a minimized set of test cases that can kill all the selected mutants. This minimized test suite is evaluated against the full set of mutants. If the mutation score obtained is greater than 99%, then the reduction is deemed to be effective. Note that we compare this score against the score of a random set of mutants of the same size, in order to handle the case where the full suite itself is not mutation adequate (or even close to adequate). Our utility answers the question: does this set of mutants better represent the test adequacy criteria represented by the full set of mutants than a random sample of the same size, and if so, by how much?

The strategy that can select the perfect set of representative mutants (the smallest set of mutants such that they have the same minimum test suite as the full set) is called the perfect strategy, with its utility denoted by U_{perfect} ⁷.

We now show how to derive an expression for the maximum U_{perfect} for the idealized system with the following restrictions.

1. We assume that we have an equal number of redundant mutants for each distinguished mutant.

From here on, we refer to a set of non-distinguished mutants as a stratum, and the entire population is referred to as the strata. Given any population of detected mutants, the mutation reduction strategy should produce a set of mutants such that if a test suite can kill all of the reduced set, the same test suite can kill all of the original mutant set (remember that T_{strategy} kills all mutants in M_{strategy}). Hence,

$$\text{kill}(T_{\text{perfect}}, M) = \text{kill}(T, M)$$

The quality of the test suite thus selected is dependent on the number of unique mutants that we are able to sample. Since we have supposed a uniform distribution, say we have x elements per stratum, and total n mutants. Our sample size s would be $p \times k$ where k is the number of strata, p is the number of samples from each stratum, and is a natural number; i.e. the sample would contain elements from each stratum, and those would have equal representation. Note that there will be at least one sample, and one strata: i.e., $s \geq 1$. Since our strata are perfectly homogeneous by construction, in practice $p = 1$ is sufficient for perfect representation, and as we shall see below, ensures maximal advantage over random sampling.

Next, we evaluate the number of different (unique) strata expected in a random sample of the same size s .

Let X_i be a random variable defined by

$$X_i = \begin{cases} 1 & \text{if strata } i \text{ appears in the sample} \\ 0 & \text{otherwise.} \end{cases}$$

Let X be the number of unique strata in the sample, which is given by: $\sum_{i=1}^k X_i$, and the expected value of X (considering that all mutants have equal chance to be sampled) is given by:

$$E(X) = E\left(\sum_{i=1}^k X_i\right) = \sum_{i=1}^k E(X_i) = k \times E(X_1)$$

Next, consider the probability that the mutant 1 has been selected, where the sample size was $s = p \times k$:

$$P[X_i = 1] = 1 - \left(\frac{k-1}{k}\right)^{pk}$$

The expectation of X_i :

$$E(X_1) = 1 \times P(X_1 = 1)$$

Hence, the expected number of unique strata appearing in a random sample is:

$$k \times E(X_1) = k - k \times \left(\frac{k-1}{k}\right)^{pk}$$

We already know that the number of unique strata appearing in each strata-based sample is k (because it is perfect, so each strata is unique). Hence, we compute the utility as the difference divided by the baseline.

$$U_{\max} = \frac{k - (k - k \times \left(\frac{k-1}{k}\right)^{pk})}{k - k \times \left(\frac{k-1}{k}\right)^{pk}} = \frac{1}{\left(\frac{k-1}{k}\right)^{pk} - 1}$$

(1)

This converges to⁸

$$\lim_{k \rightarrow \infty} \frac{1}{\left(\frac{k-1}{k}\right)^{pk} - 1} = \frac{1}{e^p - 1} \quad (2)$$

and has a maximum value when $p = 1$.

$$U_{\max} = \frac{1}{e - 1} \approx 58.2\% \quad (3)$$

Note that this is the mean improvement expected over random sampling for uniform distribution of redundant mutants in strata (and with oracular knowledge). That is, individual samples could still be arbitrarily advantageous (after all, the perfect strata sample itself is one potential random sample), but on average this is the expected gain over random samples.

How do we interpret this result? If you have a robust set of test cases that is able to uniquely identify distinguishable mutants, then given an arbitrary program, you can expect a perfect strategy to have at least a mean 58.2% advantage over random sample of the same efficiency in terms of effectiveness. However, if the program produces redundant mutants that

are skewed, then the advantage of perfect strategy with oracular knowledge will increase (depending on the amount of skew). Similarly, if the tests are not sufficient to identify distinguishable mutants uniquely, we can expect the advantage of the perfect strategy to decrease. Finally, strategies can rarely be expected to come close to perfection in terms of classifying mutants in terms of their behavior without post hoc knowledge of the kills. Hence the advantage held by such a strategy would be much much lower (or it may not even have an advantage).

Table 1: PIT Mutation Operators. The (*) operators were added or extended by us.

| | |
|-----|---|
| IN | Remove negative sign from numbers |
| RV | Mutate return values |
| M | Mutate arithmetic operators |
| VMC | Remove void method calls |
| NC | Negate conditional statements |
| CB | Modify boundaries in logical conditions |
| I | Modify increment and decrement statements |
| NMC | Remove non-void method calls, returning default value |
| CC | Replace constructor calls, returning null |
| IC | Replace inline constants with default value |
| RI* | Remove increment and decrement statements |
| EMV | Replace member variable assignments with default value |
| ES | Modify switch statements |
| RS* | Replace switch labels with default (thus removing them) |
| RC* | Replace boolean conditions with true |
| DC* | Replace boolean conditions with false |

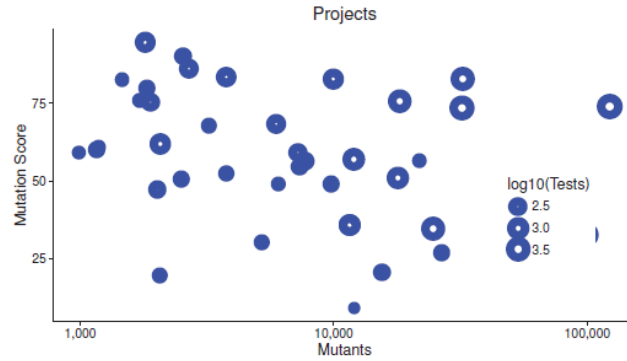


Figure 1: Distribution of number of mutants and test suites. It shows that we have a reasonable non-biased sample with both large programs with high mutation scores, and also small low scoring projects.

● 4. EMPIRICAL ANALYSIS

The above analysis provides a theoretical framework for evaluating the advantage a sampling method can have over random sampling, with a set of mutants and test suite constructed with simplifying assumptions. It also gives us an expected limit for how good these techniques could get for a uniform distribution of mutants. However, in practice, it is unlikely that real test suites and mutant sets meet our assumptions. What advantage can we expect to gain with real software systems, even if we allow our hypothetical method to make use of prior knowledge of the results of mutation analysis? To find out, we examine a large set of real-world programs and their test suites.

Our selection of sample programs for this empirical study of the limits of mutation reduction was driven by a few over-riding concerns. Our primary requirement was that our results should be as representative as possible of real-world programs. Second, we strove for a statistically significant result, therefore reducing the number of variables present in the experiments for reduction of variability due to their presence.

We chose a large random sample of Java projects from Github [22]9 and the Apache Software Foundation [6] that use the popular Maven [7] build system. From an initial 1, 800 projects, we eliminated aggregate projects, and projects without test suites, which left us with 796 projects. Out of these, 326 projects compiled (common reasons for failure included unavailable dependencies, compilation errors due to syntax, and bad configurations). Next, projects that did not pass their own test suites were eliminated since the analysis requires a passing test suite. Tests that timed out for particular mutants were assumed to have not detected the mutant. The tests that completely failed to detect any of the mutants were eliminated as well, as these were redundant to our analysis. We also removed all projects with trivial test suites, leaving only those that had at least 100 test cases. This left us with 39 projects. The projects are given in Table 2.

| | nmc | rv | ic | dc | nc | rc | vmc | cc | emv | m | cb | i | rs | es | in |
|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| nmc | 1 | 0.06 | 0.05 | 0.06 | 0.06 | 0.06 | 0.05 | 0.06 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.04 |
| rv | 0.03 | 1 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.04 | 0.04 | 0.03 |
| ic | 0.13 | 0.13 | 1 | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 | 0.1 | 0.11 | 0.11 | 0.11 | 0.11 | 0.11 | 0.12 |
| dc | 0.11 | 0.11 | 0.11 | 1 | 0.11 | 0.11 | 0.11 | 0.11 | 0.1 | 0.09 | 0.09 | 0.09 | 0.09 | 0.09 | 0.15 |
| nc | 0.05 | 0.05 | 0.05 | 0.05 | 1 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.04 | 0.04 | 0.04 | 0.04 | 0.05 |
| rc | 0.09 | 0.09 | 0.09 | 0.09 | 0.09 | 1 | 0.09 | 0.09 | 0.09 | 0.08 | 0.07 | 0.07 | 0.07 | 0.07 | 0.07 |
| vmc | 0.21 | 0.21 | 0.21 | 0.21 | 0.21 | 0.21 | 1 | 0.21 | 0.24 | 0.2 | 0.21 | 0.21 | 0.2 | 0.2 | 0.25 |
| cc | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 1 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 |
| emv | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.09 | 0.1 | 1 | 0.11 | 0.09 | 0.1 | 0.09 | 0.08 | 0.07 |
| m | 0.22 | 0.22 | 0.22 | 0.23 | 0.22 | 0.22 | 0.22 | 0.22 | 0.22 | 1 | 0.22 | 0.2 | 0.2 | 0.19 | 0.13 |
| cb | 0.23 | 0.23 | 0.23 | 0.23 | 0.23 | 0.23 | 0.22 | 0.22 | 0.23 | 0.18 | 1 | 0.18 | 0.17 | 0.17 | 0.24 |
| i | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 | 0.13 | 0.13 | 0.14 | 0.13 | 0.13 | 1 | 0.13 | 0.13 | 0.15 |
| rs | 0.32 | 0.32 | 0.32 | 0.33 | 0.32 | 0.32 | 0.32 | 0.31 | 0.31 | 0.32 | 0.32 | 0.28 | 1 | 0.28 | 0.23 |
| es | 0.26 | 0.26 | 0.26 | 0.27 | 0.26 | 0.26 | 0.27 | 0.25 | 0.27 | 0.27 | 0.26 | 0.26 | 0.26 | 1 | 0.26 |
| in | 0.36 | 0.36 | 0.36 | 0.43 | 0.36 | 0.36 | 0.36 | 0.36 | 0.36 | 0.36 | 0.38 | 0.38 | 0.34 | 0.34 | 1 |

Figure 2: Subsumption rate between operators. Note that subsumption is not a symmetrical relation. No operators come close to full subsumption. This suggests that none of the operators studied are redundant.

Table 2: The projects mutants and test suites

| Project | $ M $ | M_{killed} | M_{killed}^{uniq} | $ T $ | $ T_{min} $ |
|--------------------|--------|--------------|---------------------|-------|-------------|
| events | 1171 | 702 | 59 | 180 | 33.87 |
| annotation-cli | 992 | 589 | 110 | 109 | 38.97 |
| mercurial-plugin | 2069 | 401 | 102 | 138 | 61.77 |
| fongo | 1461 | 1209 | 175 | 113 | 70.73 |
| config-magic | 1188 | 721 | 204 | 112 | 74.55 |
| claz | 5242 | 1583 | 151 | 140 | 64.00 |
| ognl | 21852 | 12308 | 2990 | 114 | 85.43 |
| java-api-wrapper | 1715 | 1304 | 308 | 125 | 107.04 |
| webbit | 3780 | 1981 | 325 | 147 | 116.93 |
| mgwt | 12030 | 1065 | 168 | 101 | 90.65 |
| csv | 1831 | 1459 | 411 | 173 | 117.97 |
| joda-money | 2512 | 1272 | 236 | 173 | 128.48 |
| mirror | 1908 | 1440 | 532 | 301 | 201.21 |
| jdbi | 7754 | 4362 | 903 | 277 | 175.57 |
| dbutils | 2030 | 961 | 207 | 224 | 141.53 |
| cli | 2705 | 2330 | 788 | 365 | 186.24 |
| commons-math1-l10n | 6067 | 2980 | 219 | 119 | 109.02 |
| mp3agic | 7344 | 4003 | 730 | 206 | 146.79 |
| asterisk-java | 15530 | 3206 | 451 | 214 | 196.32 |
| pipes | 3216 | 2176 | 338 | 138 | 120.00 |
| hank | 26622 | 7109 | 546 | 171 | 162.88 |
| java-classmate | 2566 | 2316 | 551 | 215 | 196.57 |
| betwixt | 7213 | 4271 | 1198 | 305 | 206.35 |
| cli2 | 3759 | 3145 | 1066 | 494 | 303.86 |
| jopt-simple | 1818 | 1718 | 589 | 538 | 158.37 |
| faunus | 9801 | 4809 | 553 | 173 | 146.11 |
| beanutils2 | 2071 | 1281 | 465 | 670 | 181.00 |
| primitives | 11553 | 4125 | 1365 | 803 | 486.71 |
| sandbox-primitives | 11553 | 4125 | 1365 | 803 | 488.56 |
| validator | 5967 | 4070 | 759 | 383 | 264.35 |
| xstream | 18030 | 9163 | 1960 | 1010 | 488.25 |
| commons-codec | 9983 | 8252 | 1393 | 605 | 444.69 |
| beanutils | 12017 | 6823 | 1570 | 1143 | 556.67 |
| configuration | 18198 | 13766 | 4522 | 1772 | 1058.36 |
| collections | 24681 | 8561 | 2091 | 2241 | 938.32 |
| jfreechart | 99657 | 32456 | 4686 | 2167 | 1696.86 |
| commons-lang3 | 32323 | 26741 | 4479 | 2456 | 1998.11 |
| commons-math1 | 122484 | 90681 | 17424 | 5881 | 4009.98 |
| jodatetime | 32293 | 23796 | 6920 | 3973 | 2333.49 |

We used PIT [13] for our analysis. PIT was extended to provide operators that it was lacking [2] (accepted into main-line). We also ensured that the final operators (Table 1) were not redundant. The redundancy matrix for the full operator set is given in Figure 2. A mutant m_1 is deemed to subsume another, say m_2 if any tests that kills m_1 is guaranteed to kill m_2 . This is extended to mutation operators whereby the fraction of mutants in o_1 killed by test cases that kills all mutants in o_2 is taken as the degree of subsumption of o_1 by o_2 . The matrix shows that the maximum subsumption was just 43% — that is, none of the operators were redundant. For a detailed description of each mutation operator, please refer to the PIT documentation [14]. To remove the effects of random noise, results for each criteria were averaged over ten runs. The mutation scores along with the sizes of test suites are given in Figure 1.

It is of course possible that our results may be biased by the mutants that PIT produces, and it may be argued that the tool we use produces too many redundant mutants, and hence the results may not be applicable to a better tool that reduces the redundancy of mutants. To account for this argument, we run our experiment in two parts, with similar procedures but with different mutants. For the first part, we use the detected mutants from PIT as is, which provides us with an upper bound that a practicing tester can expect to experience, now, using an industry-accepted tool. For the second part, we choose only distinguishable mutants [3] from the original set of detected mutants. What this does is to reduce the number of samples from each stratum to 1, and hence eliminate the skew in mutant population. Note that this requires post-hoc knowledge of mutant kills (not just that the mutants produce different failures, but also that available tests in the suite can distinguish between both), and is the best one can do for the given projects to enhance the utility of any strategy against random sampling. We provide results for both the practical and more theoretically interesting distinguishable sets of mutants. Additionally, in case adequacy has an impact, we chose the projects that had plausible mutation-adequate test suites, and computed the possible advantage separately.

4.1 Experiment

Our task is to find the U_{perfect} for each project. The requirements for a perfect strategy are simple:

1. The mutants should be representative of the full set. That is,

$$\text{kill}(T_p, M) = \text{kill}(T, M)$$

2. The mutants thus selected should be non-redundant. That is,

$$\forall m \in M_p \quad \text{kill}(T_p, M_p \setminus \{m\}) \subset \text{kill}(T_p, M_p)$$

The minimal mutant set suggested by Ammann et al. [3] satisfies our requirements for a perfect strategy, since it is representative — a test suite that can kill the minimal mutants can kill the entire set of mutants — and it is non-redundant with respect to the corresponding minimal test suite.

Ammann et al. [3] observed that the cardinality of a minimal mutant set is the same as the cardinality of the corresponding minimal test suite. That is,

$$|M_{\text{perfect}}^{\min}| = |\text{MinTest}(T, M)| = |T_{\text{all}}^{\min}|$$

Finding the true minimal test suite for a set of mutants is NP-complete¹⁰. The best possible approximation algorithm is Chvatal's [12], using a greedy algorithm where each iteration

tries to choose a set that covers the largest number of mutants. This is given in Algorithm 1. In the worst case, if the number of mutants is n , and the smallest test suite that can cover it is k , this algorithm will achieve a $k \cdot \ln(n)$ approximation. We note that this algorithm is robust in practice, and usually gets results close to the actual minimum k (see Figure 3). Further, Feige [21] showed that this is the closest approximation ratio that an algorithm can reach for set cover so long as $NP \neq P$ ¹¹.

Since it is an approximation, we average the greedily estimated minimal test suite size over 100 runs. The variability is given in Figure 3, ordered by the size of minimal test suite. Note that there is very little variability, and the variability decreases as the size of test suite increases. All we need now is to find the effectiveness of random sampling for the same number of mutants as produced by the perfect strategy.

Algorithm 1 Finding the minimal test suite

```

function MINTEST( $Tests, Mutants$ )
     $T \leftarrow Tests$ 
     $M \leftarrow kill(T, Mutants)$ 
     $T_{min} \leftarrow \emptyset$ 
    while  $T \neq \emptyset \vee M \neq \emptyset$  do
         $t \leftarrow random(\max_t |kill(\{t\}, M)|)$ 
         $T \leftarrow T \setminus \{t\}$ 
         $M \leftarrow kill(T, Mutants)$ 
         $T_{min} \leftarrow T_{min} \cup \{t\}$ 
    end while
    return  $T_{min}$ 
end function
    
```

Next, we randomly sample $|M_{perfect}^{min}|$ mutants from the original set M_{random} , obtain the minimal test suite of this sample T_{random}^{min} , and find the mutants from the original set that are killed by this test suite $kill(T_{random}^{min}, M)$, which is used to compute the utility of perfect strategy with respect to that particular random sample. The experiments were repeated 100 times for each project, and averaged to compute $U_{perfect}$ for the project under consideration.

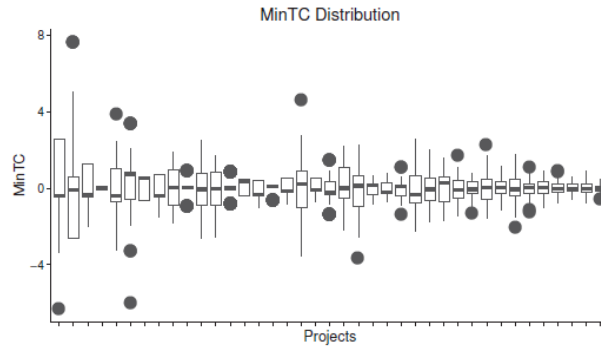


Figure 3: Variation of minimal test cases for each sample as a percentage difference from the mean ordered by mean minimal test suite size. There is very little variation, and the variation decreases with test suite size.

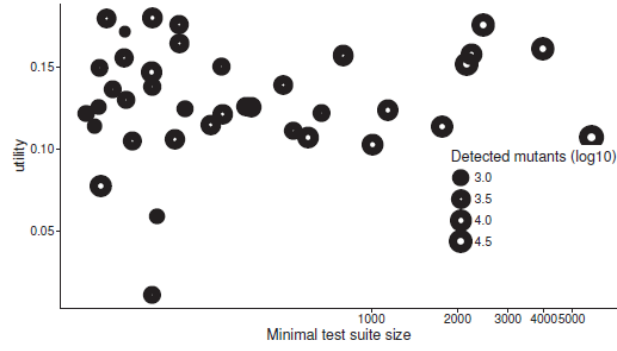


Figure 4: The figure plots utility (y-axis) against the average minimal test suite size (log10). Bubble size represents the magnitude of detected mutants (log10). The figure suggests that there is no correlation between utility and average minimal test suite size.

5. RESULTS

5.1 All Mutants

Our results are given in Table 3. We found that the largest utility achieved by the perfect strategy was 17.997%, for project faunus, while the lowest utility was 1.153%, for project joda-money. The mean utility of the perfect strategy was 13.078%. A one sample u-test suggests that 95% of projects have maximum utility between 12.218% and 14.26% ($p < 0.001$). The distribution of utility for each project is captured in Figure 6. Projects are sorted by average minimal test suite size.

One may wonder if the situation improves with either test suite size or project size. We note that the utility U_p has low correlation with total mutants, detected mutants (shown in Figure 5), mutation score, and minimal test suite size (shown in Figure 4). The correlation factors are given in Table 5.

An analysis of variance (ANOVA) to determine significant variables affecting U_{perfect} suggests that the variability due to project is a significant factor ($p < 0.001$) and interacts with $\text{kill}(T_{\text{random}}, M)$ strongly. $\mu\{U_p\} = \text{project} + \text{kill}(Tr, M) + \text{project} \times \text{kill}(Tr, M)$ The variable project has a correlation of 0.682 with the U_{perfect} , and the combined terms have a correlation of 0.9995 with U_{perfect} .

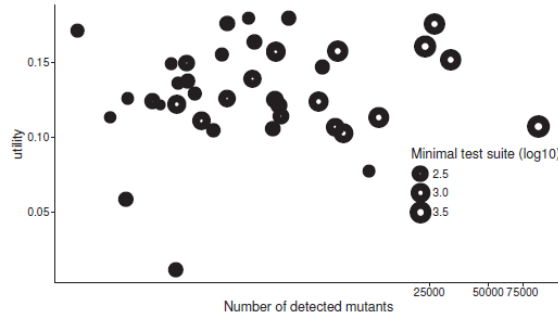


Figure 5: The figure plots utility (y-axis) against the number of detected mutants. Bubble size represents the magnitude of average minimal test suite size (log10). The figure suggests that there is no correlation between utility and number of detected mutants.

Table 5: The correlation of utility for all mutants, killed mutants, mutation score, and minimal test suite size, based on both full set of mutants, and also considering only distinguished mutants

| | R_{all}^2 | $R_{distinguished}^2$ |
|--------------|-------------|-----------------------|
| M | -0.02 | -0.03 |
| M_{kill} | -0.03 | -0.01 |
| M_{kill}/M | -0.02 | -0.00 |
| T^{min} | -0.01 | -0.02 |

Table 3: The maximum utility achievable by a perfect strategy for each project

| Project | $ kill(T, M) $ | $ kill(T_r, M) $ | U_{perf} |
|--------------------|----------------|------------------|------------|
| events | 702 | 662.97 | 0.06 |
| annotation-cli | 589 | 529.51 | 0.11 |
| mercurial-plugin | 401 | 342.91 | 0.17 |
| fongo | 1209 | 1052.99 | 0.15 |
| config-magic | 721 | 640.91 | 0.13 |
| claz | 1583 | 1402.39 | 0.13 |
| ognl | 12308 | 11426.09 | 0.08 |
| java-api-wrapper | 1304 | 1148.52 | 0.14 |
| webbit | 1981 | 1793.96 | 0.10 |
| mgwt | 1065 | 949.96 | 0.12 |
| csv | 1459 | 1282.93 | 0.14 |
| joda-money | 1272 | 1257.55 | 0.01 |
| mirror | 1440 | 1252.50 | 0.15 |
| jdbi | 4362 | 3914.73 | 0.11 |
| dbutils | 961 | 854.83 | 0.12 |
| cli | 2330 | 2069.84 | 0.13 |
| commons-math1-l10n | 2980 | 2527.66 | 0.18 |
| mp3agic | 4003 | 3620.41 | 0.11 |
| asterisk-java | 3206 | 2754.69 | 0.16 |
| pipes | 2176 | 1884.73 | 0.16 |
| hank | 7109 | 6200.08 | 0.15 |
| java-classmate | 2316 | 1969.76 | 0.18 |
| betwixt | 4271 | 3809.19 | 0.12 |
| cli2 | 3145 | 2760.66 | 0.14 |
| jopt-simple | 1718 | 1546.21 | 0.11 |
| faunus | 4809 | 4078.22 | 0.18 |
| beanutils2 | 1281 | 1141.73 | 0.12 |
| primitives | 4125 | 3565.83 | 0.16 |
| sandbox-primitives | 4125 | 3563.85 | 0.16 |
| validator | 4070 | 3616.71 | 0.13 |
| xstream | 9163 | 8307.12 | 0.10 |
| commons-codec | 8252 | 7455.50 | 0.11 |
| beanutils | 6823 | 6071.53 | 0.12 |
| configuration | 13766 | 12359.89 | 0.11 |
| collections | 8561 | 7392.63 | 0.16 |
| jfreechart | 32456 | 28171.19 | 0.15 |
| commons-lang3 | 26741 | 22742.46 | 0.18 |
| commons-math1 | 90681 | 81898.25 | 0.11 |
| jodatime | 23796 | 20491.96 | 0.16 |

Table 4: The maximum utility achievable by a perfect strategy for each project using distinguishable mutants

| Project | $ kill(T, M) $ | $ kill(T_r, M) $ | U_{perf} |
|--------------------|----------------|------------------|------------|
| events | 59 | 49.15 | 0.20 |
| annotation-cli | 110 | 93.68 | 0.18 |
| mercurial-plugin | 102 | 80.95 | 0.26 |
| fongo | 175 | 145.13 | 0.21 |
| config-magic | 204 | 171.60 | 0.19 |
| claz | 151 | 129.24 | 0.17 |
| ognl | 2990 | 2835.77 | 0.05 |
| java-api-wrapper | 308 | 259.87 | 0.19 |
| webbit | 325 | 280.89 | 0.16 |
| mgwt | 168 | 140.60 | 0.20 |
| csv | 411 | 349.30 | 0.18 |
| joda-money | 236 | 230.76 | 0.02 |
| mirror | 532 | 444.17 | 0.20 |
| jdbi | 903 | 783.99 | 0.15 |
| dbutils | 207 | 170.60 | 0.21 |
| cli | 788 | 688.05 | 0.15 |
| commons-math1-l10n | 219 | 177.86 | 0.23 |
| mp3agic | 730 | 639.01 | 0.14 |
| asterisk-java | 451 | 372.25 | 0.21 |
| pipes | 338 | 288.41 | 0.17 |
| hank | 546 | 465.52 | 0.17 |
| java-classmate | 551 | 450.46 | 0.22 |
| betwixt | 1198 | 1055.30 | 0.14 |
| cli2 | 1066 | 903.30 | 0.18 |
| jopt-simple | 589 | 514.36 | 0.15 |
| faunus | 553 | 467.03 | 0.18 |
| beanutils2 | 465 | 392.30 | 0.19 |
| primitives | 1365 | 1155.09 | 0.18 |
| sandbox-primitives | 1365 | 1155.01 | 0.18 |
| validator | 759 | 647.36 | 0.17 |
| xstream | 1960 | 1691.84 | 0.16 |
| commons-codec | 1393 | 1192.29 | 0.17 |
| beanutils | 1570 | 1341.04 | 0.17 |
| configuration | 4522 | 3934.21 | 0.15 |
| collections | 2091 | 1750.05 | 0.19 |
| jfreechart | 4686 | 3910.15 | 0.20 |
| commons-lang3 | 4479 | 3663.98 | 0.22 |
| commons-math1 | 17424 | 15139.90 | 0.15 |
| jodatime | 6920 | 5801.10 | 0.19 |

5.2 Distinguishable Mutants

Our results are given in Table 4. We found that the largest utility achieved by the perfect strategy was 26.159%, for project mercurial-plugin, while the lowest utility was 2.283%, for project joda-money.

The mean utility of the perfect strategy was 17.545%. A one sample u-test showed that 95% of projects have a maximum utility between 16.912% and 18.876% ($p < 0.001$).

The utility distribution for each project is captured in Figure 7. The projects are sorted by the average minimal test suite size.

This situation does not change with either test suite or project size.

The utility U_p has low correlation with total mutants, de-tected mutants, mutation score, and minimal test suite size. The correlation factors are given in Table 5.

An analysis of variance (ANOVA) to determine significant variables affecting $U_{perfect}$ found that the variability due to project is a significant factor ($p < 0.001$) and strongly interacts with $kill(T_{random}, M)$. $\mu\{U_p\} = \text{project} + kill(T_r, M) + \text{project} \times kill(T_r, M)$ The variable project has a correlation of 0.734 with the $U_{perfect}$, and the combined terms have a correlation of 0.9994 with $U_{perfect}$.

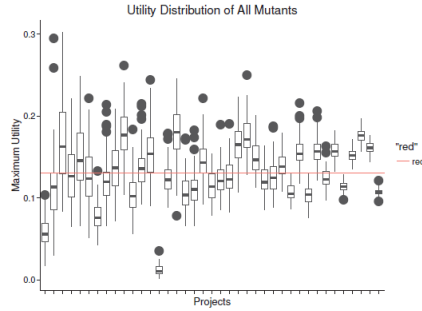


Figure 6: Using all mutants.
Distribution of mean utility using distinguished mutants across projects. The projects are ordered by the cardinality of mean minimal test suite. The red line indicates the mean of all observations.

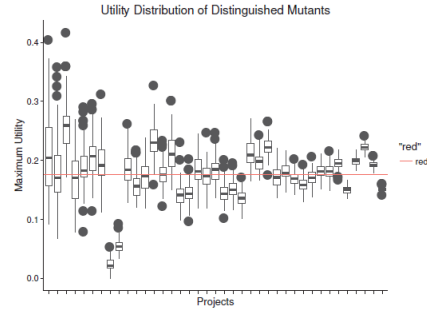


Figure 7: Using distinguished mutants.

5.3 Adequate Mutants

Finally, one may ask if adequacy has an impact on the effectiveness of selection strategies. Following the industry practice of deeming well-tested projects adequate after dis-counting equivalent mutants [47, 52–54], we chose large well tested projects that had at least 10, 000 mutants and a mutation score of at least 70% (in the range of similar studies above) which were deemed adequate. We evaluated the utility for configuration, commons-lang3, commons-math1, jodatetime and found that they have a mean maximum utility of 13.955%. These same projects have a distinguished mean maximum utility of 17.893%. This suggests that adequacy does not have a noticeable impact on the effectiveness of selection strategies.

● 6. DISCUSSION

Mutation analysis is an invaluable tool that is often difficult to use in practice due to hefty computational requirements. There is ongoing and active research to remedy this situation using different mutation reduction strategies. Hence, it is important to understand the amount by which one can hope to improve upon the simplest baseline strategy for reduction — pure random sampling.

Our theoretical analysis of a simple idealized system finds a mean improvement of 58.2% over random sampling for a mutation reduction strategy with oracular knowledge of mutation kills given a uniform distribution of mutants. This serves as an upper bound of what any known mutation reduction strategy could be expected to achieve (under the assumption that the mutant distribution is reasonably close to uniform).

Our empirical analysis using a large number of open source projects reveals that the practical limit is much lower, how-ever, on average only 13.078% for mutants produced by PIT. Even if we discount the effects of skew, by using only distinguished mutants, the potential improvement is restricted to 17.545% on average.

It is important to distinguish the different questions that the theory and empirical analysis tackle. The theoretical limit shows the best that can be done by a perfect mutation strategy given the worst distribution of mutants one may encounter. On the other hand, the empirical analysis finds the average utility of a perfect strategy without regard to the distribution of mutants in different programs. However, given that the effects of skew were found to be rather weak (only 4.467%) the theoretical bound is reasonable for the empirical question too.

The empirical upper bounds on gain in utility are surprisingly low, and call into question the effort invested into improving mutation reduction strategies. Of course, one can still point

out that random sampling is subject to the vagaries of chance, as one can get arbitrarily good or bad samples. However, our results suggest that the variance of individual samples is rather low, and the situation improves quite a bit with larger projects — e.g. the variance of commons-math1 is just 0.397%. Hence the chances for really bad samples are very low in the case of projects large enough to really need mutant reduction, and drop quickly as the number of test cases increases. One may wonder if the adequacy of test suites has an impact, but our analysis of projects with adequate test suites suggests that there is very little difference due to adequacy (Uperfect =13.955%). In general, using accepted standard practices for statistical sampling to produce reasonably-sized random mutant samples should be practically effective for avoiding unusually bad results due to random chance. The added advantage is that random sampling is easy to implement and incurs negligible overhead.

We note that our framework is applicable not only to selective mutation, but also to mutation implementors looking to add new mutators. Say a mutation implementor has a perfect set of mutation operators such that their current set of mutants does not have any redundant mutants (practically infeasible given our shallow understanding of mutant semiotics). Even if we consider the addition of a new set of random mutants that do not improve the mutation set at all, in that they are redundant with respect to the original set (rare in practice, given that we are introducing new mutants), the maximum disadvantage thus caused is bounded by our limit (18.876% upper limit for 95% of projects). However, at least a few of the new mutants can be expected to improve the representativeness of a mutation set compared to the possible faults. Since we can't bound the number of distinguishable mutants that may be introduced, there is no upper bound for the maximum advantage gained by adding new mutation operators. Adding new operators is especially attractive in light of recent results showing classes of real faults that are not coupled to any of the operators currently in common use [32].

Our previous research [25] suggests that a constant number of mutants (a theoretical maximum of 9, 604, and 1,000 in practice for 1% accuracy) is sufficient for computing mutation score with high accuracy irrespective of the total number of mutants. This suggests that sampling will lead to neither loss of effectiveness nor loss of accuracy, and hence addition of new mutation operators (and sampling the required number of mutants) is potentially a very fruitful endeavour.

● 7. THREATS TO VALIDITY

While we have taken care to ensure that our results are unbiased, and have tried to eliminate the effects of random noise. Random noise can result from non-representative choice of project, tool, or language, and can lead to skewed strata and bias in empirical result. Our results are subject to the following threats.

Threats due to approximation: We use the greedy algorithm due to Chvatal [12] for approximating the minimum test suite size. While this is guaranteed to be $H(|M|)$ approximate, there is still some scope for error. We guard against this error by taking the average of 100 runs for each observation. Secondly, we used random samples to evaluate the effectiveness of random sampling. While we have used 100 trials each for each observation, the possibility of bias does exist.

Threats due to sampling bias: To ensure representativeness of our samples, we opted to use search results from the Github repository of Java projects that use the Maven build system. We picked all projects that we could retrieve given the Github API, and selected from these only based on constraints of building and testing. However, our sample of programs could be biased by skew in the projects returned by Github.

Bias due to tool used: For our study, we relied on PIT. We have done our best to extend PIT to provide a reasonably sufficient set of mutation operators, ensuring also that the mutation operators are non-redundant. Further, we have tried to minimize the impact of redundancy by considering the effect of distinguished mutants. There is still a possibility that the kind of mutants produced may be skewed, which may impact our analysis. Hence, this study needs to be repeated with mutants from diverse tools and projects in future.

● 8. CONCLUSION

Our research suggests that blind random sampling of mutants is highly effective compared to the best achievable bound for mutation reduction strategies, using perfect knowledge of mutation analysis results, and there is surprisingly little room for improvement. Previous researchers showed that there is very little advantage to current operator selection strategies compared to random sampling [53, 54]. However, the experiment lacked direct comparison with random sampling of the same number of mutants. Secondly it was also shown that current strategies fare poorly [3] when compared to the actual minimum mutant set, but lacked comparison to random sampling. Our contribution is to show that there is a theoretical limit to the improvement that any reduction strategy can have irrespective of the intelligence of the strategy, and also a direct empirical comparison of effectiveness of the best strategy possible with random sampling.

Our theoretical investigation suggests a mean advantage of 58.2% for a perfect mutation reduction strategy with oracular knowledge of kills over random sampling given an arbitrary program, under the assumption of no skew in redundant mutants. Empirically, we find a much lower advantage 13.078% for a perfect reduction strategy with oracular knowledge. Even if we eliminate the effects of skew in redundant mutant population by considering only distinguished mutants, we find that the advantage of a perfect mutation reduction strategy is only 17.545% in comparison to random sampling. The low impact of skew (4.467%) suggests that our simplifying assumptions for theoretical analysis were not very far off the mark. The disparity between the theoretical prediction and empirical results is due to the inadequacies of real world test suites, resulting in a much smaller minimum mutant set than the distinguishable mutant set. We note that mutation reduction strategies routinely claim high reduction factors, and one might expect a similar magnitude of utility over random sampling, which fails to materialize either in theory or practice.

The second takeaway from our research is that a researcher or an implementor of mutation testing tools should consider the value of implementing a mutation reduction strategy carefully given the limited utility we observe. In fact, our research [24] suggests that popular operator selection strategies we examined have reduced utility compared to random sampling, and even strata sampling techniques based on program elements seldom exceed a 10% improvement. Given that the variability due to projects is significant, a testing practitioner would also do well to consider whether the mutation reduction strategy being used is suited for the particular system under test (perhaps based on historical data for that

project, or projects that are in some established sense similar). Random sampling of mutants is not extremely far from an empirical upper bound on an ideal mutation reduction strategy, and has the advantage of having little room for an unanticipated bias due to a “clever” selection method that might not work well for a given project. The limit re-reported here is based on using full knowledge of the mutation kill matrix, which is, to say the least, difficult to attain in practice.

Perhaps the most important takeaway from our research is that it is possible to improve the effectiveness of mutation analysis, not by removing mutation operators, but rather by further research into newer mutation operators (or new categories of mutation operators such as domain specific operators for concurrency or resource allocation). Our research suggests that the maximum reduction in utility due to addition of newer operators is just 23.268%, while there is no limit to the achievable improvement.

● REFERENCES

- [1] A. T. Acree, Jr. On Mutation. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980.
- [2] P. Ammann. Transforming mutation testing from the technology of the future into the technology of the present. In International Conference on Software Testing, Verification and Validation Workshops. IEEE, 2015.
- [3] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In International Conference on Software Testing, Verification and Validation, pages 21–30, Washington, DC, USA, 2014. IEEE Computer Society.
- [4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In International Conference on Software Engineering, pages 402–411. IEEE, 2005.
- [5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Transactions on Software Engineering, 32(8), 2006.
- [6] Apache Software Foundation. Apache commons. <http://commons.apache.org/>.
- [7] Apache Software Foundation. Apache maven project. <http://maven.apache.org>.
- [8] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for c. Software Testing, Verification and Reliability, 11(2):113–136, 2001.
- [9] T. A. Budd. Mutation Analysis of Program Test Data. PhD thesis, Yale University, New Haven, CT, USA, 1980.
- [10] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 220–233. ACM, 1980.
- [11] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. Mutation analysis. Yale University, Department of Computer Science, 1979.
- [12] V. Chvatal. A greedy heuristic for the set-covering problem. Mathematics of operations research, 4(3):233–235, 1979.

- [13] H. Coles. Pit mutation testing. <http://pitest.org/>.
- [14] H. Coles. Pit mutation testing: Mutators. <http://pitest.org/quickstart/mutators>.
- [15] M. Daran and P. Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 158–171. ACM, 1996.
- [16] R. A. DeMillo, D. S. Guindi, W. McCracken, A. Offutt, and K. King. An extended overview of the mothra software testing environment. In International Conference on Software Testing, Verification and Validation Workshops, pages 142–151. IEEE, 1988.
- [17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
- [18] L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In IEEE 6th ICST, Luxembourg, 2013.
- [19] A. Derezińska. A quality estimation of mutation clustering in c# programs. In New Results in Dependability and Computer Systems, pages 119–129. Springer, 2013.
- [20] A. Derezińska. Toward generalization of mutant clustering results in mutation testing. In Soft Computing in Computer and Information Science, pages 395–407. Springer, 2015.
- [21] U. Feige. A threshold of $\ln n$ for approximating set cover. pages 634–652, 1998.
- [22] GitHub Inc. Software repository. <http://www.github.com>.
- [23] R. Gopinath. Replication: Limits of mutation reduction strategies. <http://eecs.osuosl.org/rahul/icse2016/>.
- [24] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. Do mutation reduction strategies matter? Technical report, Oregon State University, Aug 2015. <http://hdl.handle.net/1957/56917>.
- [25] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. How hard does mutation analysis have to be, anyway? In International Symposium on Software Reliability Engineering. IEEE, 2015.
- [26] A. (<http://cstheory.stackexchange.com/users/37275/anonymous>). What is the reverse of greedy algorithm for setcover? Theoretical Computer Science Stack Exchange. url:<http://cstheory.stackexchange.com/q/33685> (version: 2016-01-29).
- [27] S. Hussain. Mutation clustering. Master’s thesis, King’s College London, Strand, London, UK, 2008.
- [28] C. Ji, Z. Chen, B. Xu, and Z. Zhao. A novel method of mutation clustering based on domain analysis. In SEKE, pages 422–425, 2009.
- [29] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In IEEE International Working Conference on Source Code Analysis and Manipulation, pages 249–258. IEEE, 2008.
- [30] Y. Jia and M. Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10):1379–1393, Oct. 2009.
- [31] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
- [32] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In ACM SIGSOFT Symposium on The Foundations of Software Engineering, pages 654–665, Hong Kong, China, 2014. ACM.

- [33] R. M. Karp. Reducibility among combinatorial problems. Springer, 1972.
- [34] B. Kurtz, P. Ammann, and J. Offutt. Static analysis of mutant subsumption. In Software Testing Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, pages 1–10, April 2015.
- [35] R. Kurtz, P. Ammann, M. Delamaro, J. Offutt, and L. Deng. Mutant subsumption graphs. In Workshop on Mutation Analysis, 2014.
- [36] R. J. Lipton. Fault diagnosis of computer programs. Technical report, Carnegie Mellon Univ., 1971.
- [37] Y.-S. Ma and S.-W. Kim. Mutation testing cost reduction by clustering overlapped mutants. Journal of Systems and Software, pages –, 2016.
- [38] A. Mathur. Performance, effectiveness, and reliability issues in software testing. In Annual International Computer Software and Applications Conference, COMPSAC, pages 604–605, 1991.
- [39] A. P. Mathur. Foundations of Software Testing. Addison-Wesley, 2012.
- [40] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. Software Testing, Verification and Reliability, 4(1):9–31, 1994.
- [41] E. S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing, Verification and Reliability, 9(4):205–232, 1999.
- [42] A. S. Namin and J. H. Andrews. Finding sufficient mutation operators via variable reduction. In Workshop on Mutation Analysis, page 5, 2006.
- [43] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In International Conference on Software Engineering, pages 100–107. IEEE Computer Society Press, 1993.
- [44] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In Mutation testing for the new century, pages 34–44. Springer, 2001.
- [45] A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, 1996.
- [46] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In ACM SIGSOFT Symposium on The Foundations of Software Engineering, pages 297–298, Aug. 2009.
- [47] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In International Conference on Software Engineering, pages 351–360. ACM, 2008.
- [48] J. Strug and B. Strug. Machine learning approach in mutation testing. In Testing Software and Systems, pages 200–214. Springer, 2012.
- [49] R. H. Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In Annual Southeast Regional Conference, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM.
- [50] W. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. Journal of Systems and Software, 31(3):185 – 196, 1995.

- [51] W. E. Wong. On Mutation and Data Flow. PhD thesis, Purdue University, West Lafayette, IN, USA, 1993. UMI Order No. GAX94-20921.
- [52] J. Zhang, M. Zhu, D. Hao, and L. Zhang. An empirical study on the scalability of selective mutation testing. In International Symposium on Software Reliability Engineering. ACM, 2014.
- [53] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In IEEE/ACM Automated Software Engineering. ACM, 2013.
- [54] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In International Conference on Software Engineering, NY, USA, 2010. ACM.

附录B 外文文献译文

关于变异约简策略的界限

Rahul Gopinath Oregon State University gopinath@eecs.orst.edu

Mohammad Amin Alipour Oregon State University alipour@eecs.orst.edu

Iftexhar Ahmed Oregon State University ahmedi@onid.orst.edu

Carlos Jensen Oregon State University cjensen@eecs.orst.edu

Alex Groce Oregon State University agroce@gmail.com

摘要

变异分析被认为是评估测试用例集有效性的最好方式，然而高昂的计算成本往往限制了它的使用。为了解决这个问题，各种变异约简策略已经被提出，旨在减少变异体的数量，同时保持详尽的变异分析的代表性。现有研究主要集中在约简策略的实现，然而在选择代表性的变异体时，这些约简策略的有效性和界限至今还没有从理论和实证的角度进行探究。

我们研究变异约简策略有效性的实际界限，并且提供一个思考绝对极限的简单理论框架。我们的结果表明，通过对真实世界开源项目随机采样，变异约简策略的有效性改进比例极限仅为 13.078%。有趣的是，通过添加新的变异算子引起的改进比例没有界限。

考虑到这是用最好最先进知识杀死的变异体的最大数量，实际操作中可能非常糟糕。我们推论出，应该将更多的精力放在改进变异分析上，而不是为了不确定的优点以选择性变异的名义移除变异算子。

目录和项目描述

关键词：软件测试，静态分析，理论分析，变异分析

1.引言

软件质量是软件行业迫切关注的问题，通常通过综合测试决定。然而，测试是用他们自己的程序，（通常）是由人编写的，它们的质量需要被监控以确保它们确实可以保证软件质量。（例如，确定测试程序是否也是一个优质的软件系统是非常重要的）。

变异分析[11,36]是目前评估测试用例集有效性的推荐方法[5]。通过引入小的句法改变对程序进行系统的转换，每一个句法改变都通过给定的测试用例进行评估。如果变异体可以被测试用例集区分出来，则认为变异体被该测

试用例集杀死。测试用例集杀死的变异体数与变异体集合的比值称为变异(杀死)得分,可以用来评估测试用例集的有效性。

变异分析在过去已经被验证了很多次。如 Andrews 等人[4.5]及最近的 Just 等人[32]的研究发现,通过变异分析产生的错误与真实的错误类似,它们的检测度和真实错误相似,最重要的是,针对变异体和真实错误的测试用例集的有效性也相似。

然而,由于变异分析需要巨大的计算资源——仅是为了获得完备的一阶变异体(一次进行一处语法改变),产生的变异体数量就需要是待测程序的数倍,而且每一个变异体都需要执行一个可能完备的测试用例集。为了解决变异分析的计算开销,大量的方法已经被提出了。这些方法被分为独立的三类:更快、更智能和更少。分别对应改进单个变异体的执行速度、变异体评估并行化和减少评估的变异体的数量。

大量的 **do fewer** 策略——旨在智能选择一个更小,更具有代表性的变异体集合的变异体约简策略——在过去已经被研究了。它们大体上可以分为变异算子选择策略,旨在寻找能够生成最有用变异体的最小变异算子子集[43,46];分层抽样[1,9]技术,识别具有高相似度的变异体组,减少变异体的数量同时保证变异体的代表性和多样性[53,54]。采用聚类[19,37]、静态分析[28,34]和其他智能技术[48]的更复杂的方法也在积极研究中[20]。

这些努力提出了一个重要的问题:对于给定的任意程序,以随机选择策略为基准,什么是一个完美的变异约简策略的实际有效性?

我们将一个选择技术的效率定义为可实现的约简数量,将有效性定义为选择代表性约简变异体集合的能力,所选择的变异体集合和原始变异体集合相比,需要相同数量的测试用例集才能杀死。技术效用是该技术有效性与随机选取有效性的比值。

我们从两个方向处理这些问题。首先,使用几个简单的假设,我们考虑一个简单的理论框架去评估最好的变异约简策略有效性的提高,并且给出变异体被杀死的预期知识。这样可以设定基准。第二,采用大量项目实证评估最好的变异约简策略可能性,给出事后(即,预言性)检测知识。就一般项目特点而言,这给了我们实际的(和乐观的)界限。

我们的贡献如下:

- 我们发现,对于均匀分布的变异体,变异约简策略有效性的理论上限为 58.2%——这种分布最利于随机抽样。随后我们表示,对于真实世界的程序,其分布的影响是很小的(4.467%),说明均匀分布是一个合理的近似。

- 通过对大量开源项目的评估，我们发现了有效性的经验上限，平均最大的实际效用是 13.078%，并且对于 95% 的项目来说，最大效用介于 12.218% 和 14.26% 之间（一个样本 $u\text{-test } P < 0.001$ ）。
- 我们发现，即使我们认为一组变异体至少被一个测试识别（因此降低了冗余变异体的扭曲影响），我们可以认为最大效用平均为 17.545%，并且对于 95% 的项目而言，最大效用介于 16.912% 和 18.876% 之间（一个样本 $u\text{-test } P < 0.001$ ）。

我们的结果对变异约简策略的未来意味着什么呢？通过随机抽样获得的优势的确是优势，但是优势小。然而，我们对变异符号的理解是不完善和不充分的，因此不能推断出这类选择方式是否有利。事实上，我们目前的研究 [24] 表明，目前的变异算子选择策略和随机抽样相比几乎没有任何优势，而且甚至基于程序元素的分层抽样，和纯随机抽样相比，取得的优势从未超过 10%。我们的结果表明，在潜在的最大效用下，应该仔细权衡改善变异体选择机制付出的努力，而且通过偏差抽样，与之相关的风险事实上使事情变得更糟。

我们也有必要进一步研究新的变异算子。研究表明，增加新的变异算子，然后随机抽取和原始变异体集合相同数量的变异体，仅仅满足了一个相似的最大劣势（ $\frac{0.189 \times 100}{1 - 0.189} = 23.268\%$ 95% 的项目上限），然而由于有效性的增加，优势基本没有上限。

变异算子移除和变异算子添加获得的改进的不对称是由抽取的随机比较样本的群组差异造成的。对于变异算子选择来说，移除变异算子后保留的完备集是原始群组的子集。因为是从原始群组中随机抽样，所以完备集可能包含来自每一层的变异体。对于变异算子添加来说，新的完备集是原始群组的超集，包含的新的变异体数和新的层数相同（新层的数量没有边界）。因为是从原始的群组随机抽样，所以不包含新增加的层级。

我们的结果表明，寻找更新的变异类别比试图减少现有的变异算子更有成效。

为了容易复制，我们使用 Knitr 组织和复现我们的研究。本文的原始 Knitr 源和 R 数据使得写这篇文章很有必要，而且做这些研究的说明是可获取到的。

2.相关工作

根据 Mathur[39]，变异分析的观点最早由 Richard Lipton 提出，由 DeMillo 等人[17]正式建立。变异分析的实际实现是由 Budd 等人[10]于 1980 年完成的。

变异分析包括不同的覆盖方法[9,40,45]；就错误产生[15]和检测难易[4,5]

而言,产生的错误和真实错误类似。Just 等人[32]使用 357 个真实缺陷研究了变异得分和测试用例集有效性之间的关系,然后发现 75%的测试用例集随着有效性的增加变异得分也随之增加,比结构覆盖的 46%更好。

由于全面分析需要执行大量的测试用例,因此通常变异分析非常昂贵[31]。有几种方法可以降低变异分析的代价,Offutt 和 Untch[44]将其分为如下类别:更少,更智能和更快。更少方法包括选择变异和变异体采样,而弱变异,变异分析并行化和空间/时间权衡属于更智能方法。最后,更快方法包括变异模式生成,代码修补和其他的一些方法。

仅仅使用变异体的子集的想法是随着变异分析出现的。Budd[9]和 Acree[1]表示,甚至抽取 10%变异体就可以 99%的接近变异得分。Mathur[38], Wong 等人[50,51]和 Offutt 等人[43]使用 Fortran 的 Mothra[16]进一步探究了这个观点。

大量的研究已经看到了变异算子选择和随机采样标准的相对优势。Wong 等人[50]比较了从每一种变异体类型中选择 $x\%$ 与仅使用两个变异算子的变异算子选择两种方式,发现这两种方式取得了相似的精度和约简度(80%)。Mresa 等人[41]使用检测代价作为变异算子选择方式。他们发现,如果需要很高的变异得分(接近 100%), $x\%$ 选择变异优于变异算子选择,并且相反地,如果需要较低的变异得分,在考虑检测变异体的代价的情况下,变异算子选择会更好。

Zhang 等人[54]将基于变异算子的变异体选择技术与随机抽样做了对比。他们发现,没有任何一个选择技术优于随机抽样。他们也发现,对较大的程序,均匀抽样比变异算子分层抽样更有效,对较小的程序则反之。最近,Zhang 等人[53]证实了,为与变异得分有非常高的相关性(99%),取样少至变异体的 5%就足够了,甚至更少的变异体更有潜力保持更高的精度。他们研究了基于变异算子的变异体选择策略的前 8 种采样策略,然后发现基于程序组件(尤其是方法)的采样策略效果最好。

一些研究试图找到一组充分的变异算子,能够降低变异的成本但是保持与变异得分的相关性。Offutt 等人[43]提出了一种 n -选择方法,一步步的移除产生变异体数量最多的变异算子。Barbosa 等人提供了一种准则来选择这种变异算子。Namin 等人[42,47]把这个问题划分为变量约简问题,然后发现 Proteum 的 108 种变异算子,仅仅其中 28 种就足够产生精确的结果。

仅仅使用声明删除算子最早由 Untch[49]提出,他发现与其他的变异算子选择方法相比,尽管这种方式生成的变异体数量最少,但与变异得分有最高的相关性($R^2 = 0.97$)。这个观点被 Deng 等人[18]进一步强化,他们定义了针

对不同语言元素的删除操作，并且发现当减少 80% 的变异体时，获得了 92% 的精度。

一个类似的变异约简策略是把相似的变异聚类在一起[20,27]，这种基于域分析[28]和基于图形学的机器学习技术[48]的方式已经被尝试了。

在变异算子和变异体包含中，不显著异于其他的变异算子或变异体被淘汰。Kurtz 等人[35]发现，高达 24 倍的约简可以通过单独使用包含关系实现，即使这个结果是基于单一程序 cal 的研究。变异体包含的研究也包括高阶变异体（HOM），在高阶变异体中，多处变异被引入同一个变异体集，通过包含组合变异体减少了单个变异体的数量。HOMs 被 Jia 等人[29,30]研究，他们发现它们可以减少 50% 的变异体。

Ammann 等人[3]发现，与最小测试用例集相对应的最小变异体集和测试用例集有相同的基数，并且提供了一种简单的算法用于寻找最小测试用例集和与之相对应的最小变异体集。他们的工作也表明这个最小的变异体集可以作为一种评估变异约简策略质量的方法。最后，Ammann 等人也发现，检查的特殊策略用于选择代表性的变异体时相当差。我们的工作是在 Ammann 等人[3]的延伸，在此基础上我们提供一个改进量的理论和经验界限，这个界限可以适用于任意变异约简策略。

与之前工作相比，我们的分析得到了理论证实，并且将选择限制与随机抽样做了对比。即，使用静态分析，我们的研究结果可以应用于聚类技术，甚至可以改进分层抽样技术。进一步，我们是第一批评估不充分测试用例集有效性的（Zhang 等人[53]仅评估不充分测试用例集的预测能力，而不是有效性）。最后，之前的研究并没有比较对于采样和变异算子选择而言，相同数量变异体的有效性，但是变异算子选择和采样的增加尺寸是相当不同的，例如 5%，10% 等。我们相信，相关人员将会对比较相同数量变异体获得的有效性更加感兴趣。

3.理论分析

对于变异约简策略理想的结果是找到可以代表整个变异体集的最小变异体集。变异约简策略通过识别冗余变异体，并将它们组合在一起，使单一变异体足以代表整个组。和随机抽样相比，这种策略的优势取决于变异体群组的特点。首先，取决于每个这样的变异体组中冗余变异体的数量。当这些群组有相同数量的变异体时（均匀分布），随机抽样的效果最好，然而对于其他的变异体分布（偏差），随机抽样的有效性则较低。然而，这种分布依赖于被评估的程序。因为我们的目标是找到任意程序的最好策略的平均优势，所以我们使用保守的变异体分布（均匀）进行理论分析（我们随后会展示，

这种偏差对真实世界变异体的实际影响少于 5%)。

接下来需要考虑的是能代表整个变异体组的最小变异体数量。如果一个变异体可以被检测它的测试集区别于另一个变异体，我们就认为就他们代表的故障，两者是可以互相区分的，然后我们从每一个不可区分的变异体集中挑选有代表性的。注意，在真实世界，可区分变异体组往往比，挑选能够杀死整个变异体组的最小测试用例集需要的最小变异体数量更大。这是因为虽然一些变异体可以被检测它们的测试用例区分，但是可能没有任何测试可以唯一杀死它们。因为对变异体组这是外部的，而且也因为这种最小变异体集不能完全代表原始群组（我们可以用一个较低数量代替，仅因为测试用例集不完备），我们假定可区分变异体可被测试用例唯一识别。然而我们注意到，不完备的测试用例集有利于随机抽样，而且降低了最好变异约简策略的优势，因为随机抽样现在可以没有任何影响的遗漏变异体。我们使用可能的最好策略推导这个系统的变异约简界限，给出变异体杀死的预期知识。

参数偏差的影响：

偏差：偏差的存在降低随机抽样的有效性，从而增加最好策略的效用。

区分性：任何不是由策略选择的可区分变异体（由于没有唯一检测的测试用例）会降低选择策略的有效性和效用。

在建立变异约简策略效用的理论框架之前，我们必须定义与原始变异体集、削减变异体集和相关测试用例集相关的一些术语。

术语： 分别用 M 和 $M_{strategy}$ 代表原始变异体集和削减变异体集。 M 中变异体被测试用例 T 杀死表示为（我们使用 M_{killed} 作为 $kill(T, M)$ 的别名）。类似的， T 中测试用例杀死 M 中变异体表示为 $cover(T, M)$ 。

$$kill : T \times M \rightarrow M$$

$$cover : T \times M \rightarrow T$$

测试用例 $T_{strategy}$ 可以杀死 $M_{strategy}$ 中的所有变异体。即， $kill(T_{strategy}, M_{strategy}) = M_{strategy}$ 。如果是被最小化的策略变异体，我们表示为 $T_{strategy}^{\min}$ 。

两个变异体 m 和 m' ，如果被不同的测试用例杀死，则可区分，记为： $cover(T, \{m\}) \neq cover(T, \{m'\})$ 。

我们使用 M_{killed}^{uniq} 表示原始集合中的可区分变异体集，如 $\forall_{m, m' \in M} cover(T, \{m\}) \neq cover(T, \{m'\})$ 。

一个策略的效用（ $U_{strategy}$ ）是和基准相比，使用该策略带来的有效性的

提高（基准是相同数目变异体的随机抽样）。即，

$$U_{strategy} = \frac{\left| kill(T_{strategy}^{\min}, M) \right|}{\left| kill(T_{random}^{\min}, M) \right|} - 1$$

注意， $T_{strategy}^{\min}$ 表示该策略选择的变异体的最小化，它适用于 $kill(T_{strategy}^{\min}, M)$ 中的全部变异体集（ M ）。

这遵从了传统的有效性评估，内容如下：从原始变异体集合，根据策略选择一个变异体子集。然后选择一个可以杀死所有选定的变异体的最小测试用例集。这个最小测试用例集用全部的变异体集进行评估。如果获得的变异得分高于 99%，那么约简被认为有效。注意，我们将这个得分与相同大小的随机变异体集的得分进行比较，为了解决完整用例集本身不是变异充分的情况（或者接近充分）。我们的效用回答这个问题：这个变异体集，是否能够比相同大小的随机抽样变异体集，更好的代表由全部变异体集代表的测试充分性准则。如果可以，好多少？

这种可以选择完备代表性变异体集（最小变异体集，和全部变异体集有相同的最小测试用例集）的策略被称为完备策略，它的效用使用 $U_{perfect}$ 表示。

我们现在展示如何用下面的限制，推导出理想化系统的最大 $U_{perfect}$ 表达式。

1. 我们假定，对于每一个可区分变异体，我们有相同数量的冗余变异体。

从这里开始，我们把一个不可区分变异体集作为 **stratum**，把整个组作为 **strata**。给出任意检测变异体组，变异约简策略应该产生一个变异体集，这样如果测试用例可以杀死所有被约简的集合，相同的测试用例可以杀死所有的原始变异体集（记住 $T_{strategy}$ 杀死 $M_{strategy}$ 中的所有变异体）。因此，

$$kill(T_{perfect}, M) = kill(T, M)$$

被选中的测试用例集的质量取决于我们能够抽样的唯一变异体的数量。既然我们已经假定是均匀分布，假设每层有 x 个元素，总共 n 个变异体。我们的抽样大小 s 应该为 $p \times k$ ，其中 k 是层数， p 是来着每层的样本数，是一个自然数。即，样本应该包含来自每层的元素，而且这些元素有相同的代表性。注意，至少要有一个样本和一层：

即 $s \geq 1$ 。由于我们的 **strata** 结构完全均匀，因此实践中用 $p=1$ 代替即可，而且正如下面我们看到的，可以保证随机抽样的最大优势。

接下来，我们评估相同 s 大小随机样本预期的不同的（唯一的）层数。

X_i 是一个随机变量，定义如下：

$$X_i = \begin{cases} 1 & \text{if strata } i \text{ appears in the sample} \\ 0 & \text{otherwise.} \end{cases}$$

X 表示样本中的唯一层的数目，由 $X = \sum_{i=1}^k X_i$ 给出， X 的期望值（认为所有变异体被抽到的机率相同）由下式给出：

$$E(X) = E\left(\sum_{i=1}^k X_i\right) = \sum_{i=1}^k E(X_i) = k \times E(X_1)$$

接下来，考虑变异体 1 被选中的概率，样本大小为 $s = p \times k$ ：

$$P[X_i = 1] = 1 - \left(\frac{k-1}{k}\right)^{pk}$$

X_i 的期望为：

$$E(X_1) = 1 \times P(X_i = 1)$$

因此，随机抽样的唯一 strata 的预期数为：

$$k \times E(X_1) = k - k \times \left(\frac{k-1}{k}\right)^{pk}$$

我们已经知道在每一个基于 strata 的样本中，唯一 strata 的数目为 k （因它是完备的，因此每一个 strata 是唯一的）。因此，我们用基准除以不同计算效用。

$$U_{\max} = \frac{k - (k - k \times \left(\frac{k-1}{k}\right)^{pk})}{k - k \times \left(\frac{k-1}{k}\right)^{pk}} = \frac{1}{\left(\frac{k}{k-1}\right)^{pk} - 1} \quad (1)$$

这收敛到

$$\lim_{k \rightarrow \infty} \frac{1}{\left(\frac{k}{k-1}\right)^{pk} - 1} = \frac{1}{e^p - 1} \quad (2)$$

当 $p=1$ 时取得最大值。

$$U_{\max} = \frac{1}{e - 1} \approx 58.2\% \quad (3)$$

注意，这是在每层均匀分布的冗余变异体中随机抽样的预期平均改进效果（和预言性知识）。即，单个样本仍然是均匀有利的（毕竟，最好的分层样本本身是一个潜在的随机抽样），但是就平均而言，这是随机抽样的预期效果。

我们怎样解释这个结果呢？如果你有一个强大的测试用例集，能够唯一识别可区分变异体，然后给出一个任意程序，通过相同效率的随机抽样，你

可以期待一个完备策略能够至少有一个平均 58.2% 的有效性优势。然而，如果程序产生有偏差的冗余变异体，那么带有预言性知识的完备策略的优势将会增加（取决于偏差的数量）。相似的，如果这些测试不足以唯一识别可区分变异体，我们可以认为完备策略的优势将会减少。最后，在没有变异体行为的事后杀死知识的情况下，就变异体分类而言，策略很少被认为能够接近完美。因此这样一个策略持有的优势将会大大大大降低（或者甚至可能没有优势）。

Table 1: PIT Mutation Operators. The (*) operators were added or extended by us.

| | |
|-----|---|
| IN | Remove negative sign from numbers |
| RV | Mutate return values |
| M | Mutate arithmetic operators |
| VMC | Remove void method calls |
| NC | Negate conditional statements |
| CB | Modify boundaries in logical conditions |
| I | Modify increment and decrement statements |
| NMC | Remove non-void method calls, returning default value |
| CC | Replace constructor calls, returning null |
| IC | Replace inline constants with default value |
| RI* | Remove increment and decrement statements |
| EMV | Replace member variable assignments with default value |
| ES | Modify switch statements |
| RS* | Replace switch labels with default (thus removing them) |
| RC* | Replace boolean conditions with true |
| DC* | Replace boolean conditions with false |

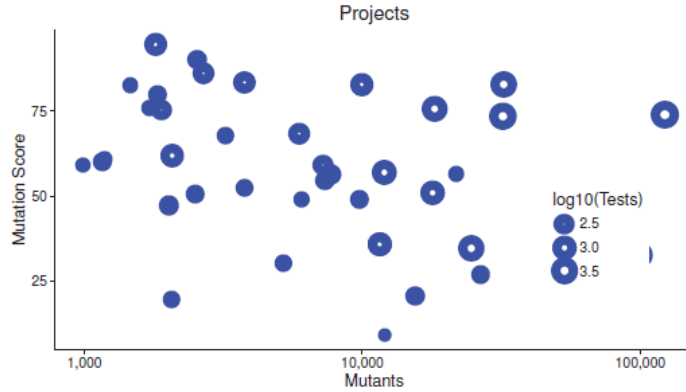


Figure 1: Distribution of number of mutants and test suites. It shows that we have a reasonable non-biased sample with both large programs with high mutation scores, and also small low scoring projects.

4.实证分析

以上的分析为用简单化假设构造的变异体集和测试用例集来评估一个采样方法和随机抽样相比可能具有的优势提供了一个理论框架。这也给了我们一个预期限制，对均匀分布的变异体，这些技术可以有多好。然后，在实践中，真实的测试用例集和变异体集是不可能满足我们的假设的。我们能够从真实软件系统中获得什么优势，即使我们允许我们的假设方法利用变异分析

结果的现有知识？为了找到答案，我们研究了大量的真实世界程序集和它们的测试用例集。

这次变异约简限制实证研究的样本程序的挑选是由几个非常重要的因素驱动的。我们的主要要求是，我们的结果应该尽可能地代表真实世界程序。其次，我们力求一个统计性的有意义的结果，因此减少变量约简实验中变量的数量。

| | nmc | rv | ic | dc | nc | rc | vmc | cc | emv | m | cb | i | ri | rs | es | in |
|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| nmc | 1 | 0.06 | 0.05 | 0.06 | 0.06 | 0.06 | 0.05 | 0.06 | 0.06 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.04 |
| rv | 0.03 | 1 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 | 0.04 | 0.04 | 0.03 |
| ic | 0.13 | 0.13 | 1 | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 | 0.1 | 0.11 | 0.11 | 0.11 | 0.11 | 0.11 | 0.11 | 0.12 |
| dc | 0.11 | 0.11 | 0.11 | 1 | 0.11 | 0.11 | 0.11 | 0.11 | 0.1 | 0.09 | 0.09 | 0.09 | 0.09 | 0.09 | 0.09 | 0.15 |
| nc | 0.05 | 0.05 | 0.05 | 0.05 | 1 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.04 | 0.04 | 0.04 | 0.04 | 0.05 |
| rc | 0.09 | 0.09 | 0.09 | 0.09 | 0.09 | 1 | 0.09 | 0.09 | 0.09 | 0.09 | 0.08 | 0.07 | 0.07 | 0.07 | 0.07 | 0.07 |
| vmc | 0.21 | 0.21 | 0.21 | 0.21 | 0.21 | 0.21 | 1 | 0.21 | 0.21 | 0.24 | 0.2 | 0.21 | 0.21 | 0.2 | 0.2 | 0.25 |
| cc | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 1 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 |
| emv | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.09 | 1 | 0.11 | 0.09 | 0.1 | 0.09 | 0.08 | 0.08 | 0.07 |
| m | 0.22 | 0.22 | 0.22 | 0.23 | 0.22 | 0.22 | 0.22 | 0.23 | 0.22 | 1 | 0.22 | 0.2 | 0.2 | 0.19 | 0.19 | 0.13 |
| cb | 0.23 | 0.23 | 0.23 | 0.23 | 0.23 | 0.23 | 0.22 | 0.22 | 0.22 | 0.23 | 1 | 0.18 | 0.18 | 0.17 | 0.17 | 0.24 |
| i | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 | 0.13 | 0.13 | 0.14 | 1 | 0.13 | 0.13 | 0.13 | 0.15 |
| ri | 0.32 | 0.32 | 0.32 | 0.33 | 0.32 | 0.32 | 0.32 | 0.31 | 0.31 | 0.32 | 0.32 | 0.32 | 1 | 0.28 | 0.28 | 0.23 |
| rs | 0.26 | 0.26 | 0.26 | 0.27 | 0.26 | 0.26 | 0.27 | 0.25 | 0.27 | 0.26 | 0.26 | 0.26 | 0.26 | 1 | 0.26 | 0.14 |
| es | 0.15 | 0.15 | 0.15 | 0.15 | 0.15 | 0.15 | 0.15 | 0.15 | 0.15 | 0.16 | 0.14 | 0.14 | 0.14 | 0.15 | 1 | 0.1 |
| in | 0.36 | 0.36 | 0.36 | 0.43 | 0.36 | 0.36 | 0.36 | 0.36 | 0.36 | 0.36 | 0.36 | 0.38 | 0.38 | 0.34 | 0.34 | 1 |

Figure 2: Subsumption rate between operators. Note that subsumption is not a symmetrical relation. No operators come close to full subsumption. This suggests that none of the operators studied are redundant.

我们从 Github[22]和使用流行 Maven[7]构建系统的 Apache 软件基金会[6]上随机选择了一个大型的 Java 项目。最初有 1800 个项目，我们消除了混合项目和没有测试用例的项目，最后留下 796 个项目。其中，326 个项目被编译了（失败的常见原因包括不可用的依赖关系，由于语法和糟糕的配置造成的编译错误）。接下来，未能通过它们自己的测试用例集的项目被舍弃了，因为分析需要一个通过的测试用例集。对特殊变异体，测试发生超时，则被假定为没有检测到该变异体。完全不能检测到任何变异体的测试用例也被舍弃了，因为这些对我们的分析是冗余的。我们也去除了测试用例集毫无价值的项目，仅仅留下那些至少有 100 个测试用例集的项目。最后留下了 39 个项目。Table 2 给出了这些项目。

Table 2: The projects mutants and test suites

| <i>Project</i> | $ M $ | M_{killed} | M_{killed}^{uniq} | $ T $ | $ T_{min} $ |
|--------------------|--------|--------------|---------------------|-------|-------------|
| events | 1171 | 702 | 59 | 180 | 33.87 |
| annotation-cli | 992 | 589 | 110 | 109 | 38.97 |
| mercurial-plugin | 2069 | 401 | 102 | 138 | 61.77 |
| fongo | 1461 | 1209 | 175 | 113 | 70.73 |
| config-magic | 1188 | 721 | 204 | 112 | 74.55 |
| clazz | 5242 | 1583 | 151 | 140 | 64.00 |
| ognl | 21852 | 12308 | 2990 | 114 | 85.43 |
| java-api-wrapper | 1715 | 1304 | 308 | 125 | 107.04 |
| webbit | 3780 | 1981 | 325 | 147 | 116.93 |
| mgwt | 12030 | 1065 | 168 | 101 | 90.65 |
| csv | 1831 | 1459 | 411 | 173 | 117.97 |
| joda-money | 2512 | 1272 | 236 | 173 | 128.48 |
| mirror | 1908 | 1440 | 532 | 301 | 201.21 |
| jdbi | 7754 | 4362 | 903 | 277 | 175.57 |
| dbutils | 2030 | 961 | 207 | 224 | 141.53 |
| cli | 2705 | 2330 | 788 | 365 | 186.24 |
| commons-math1-l10n | 6067 | 2980 | 219 | 119 | 109.02 |
| mp3agic | 7344 | 4003 | 730 | 206 | 146.79 |
| asterisk-java | 15530 | 3206 | 451 | 214 | 196.32 |
| pipes | 3216 | 2176 | 338 | 138 | 120.00 |
| hank | 26622 | 7109 | 546 | 171 | 162.88 |
| java-classmate | 2566 | 2316 | 551 | 215 | 196.57 |
| betwixt | 7213 | 4271 | 1198 | 305 | 206.35 |
| cli2 | 3759 | 3145 | 1066 | 494 | 303.86 |
| jopt-simple | 1818 | 1718 | 589 | 538 | 158.37 |
| faunus | 9801 | 4809 | 553 | 173 | 146.11 |
| beanutils2 | 2071 | 1281 | 465 | 670 | 181.00 |
| primitives | 11553 | 4125 | 1365 | 803 | 486.71 |
| sandbox-primitives | 11553 | 4125 | 1365 | 803 | 488.56 |
| validator | 5967 | 4070 | 759 | 383 | 264.35 |
| xstream | 18030 | 9163 | 1960 | 1010 | 488.25 |
| commons-codec | 9983 | 8252 | 1393 | 605 | 444.69 |
| beanutils | 12017 | 6823 | 1570 | 1143 | 556.67 |
| configuration | 18198 | 13766 | 4522 | 1772 | 1058.36 |
| collections | 24681 | 8561 | 2091 | 2241 | 938.32 |
| jfreechart | 99657 | 32456 | 4686 | 2167 | 1696.86 |
| commons-lang3 | 32323 | 26741 | 4479 | 2456 | 1998.11 |
| commons-math1 | 122484 | 90681 | 17424 | 5881 | 4009.98 |
| jodatetime | 32293 | 23796 | 6920 | 3973 | 2333.49 |

我们使用 PIT 进行我们的分析。PIT 被扩展后用于提供它缺少的变异算子[2]（接受为主线）。我们也保证最终的变异算子（Table 1）不是冗余的。图 2 给出了完备算子集的冗余矩阵。一个变异体 m_1 被认为包含另一个变异体 m_2 ，如果任何可以杀死 m_1 的测试用例都一定可以杀死 m_2 。这个被延伸到变异算子上，能够杀死 o_2 中全部变异体的测试用例也能杀死 o_1 中的变异体，则认为 o_1 被 o_2 包含。矩阵表明最大包含仅为 43%——即，没有任何一个变异算子是冗余的。每个变异算子的具体描述，请参照 PIT 文档[14]。为了去除随机噪音的影响，对每个标准结果都平均进行了 10 次实验。图 1 给出了变异得分随测试用例集大小的变化。

当然使用 PIT 产生的变异体，我们的结果可能是有偏差的，而且结果可能是有争议的，因为我们使用的工具产生了太多的冗余变异体，因此结果可能不适用于冗余性降低的更好工具。为了解决这个争议，我们分两部分进行我们的实验，步骤相似但变异体不同。对第一部分，我们使用 PIT 的测试变

异体, PIT 可以给我们提供一个实际测试人员可以体验到的上限, 现在, 用一种业内可接受的工具。对第二部分, 我们仅选择来自原始测试变异体集的可区分变异体[3]。这是为了将每层样本的数量降为 1, 因此消除变异体组中的偏差。注意, 这个需要杀死变异体的后事知识 (不仅是变异体产生不同的故障, 而且可用的测试用例可以在两者间区分), 而且对于给定的工程, 这是能采取的保证任何较随机抽样而言的策略的效用的最好方式。我们为实际的和更理论化的有趣可区分变异体集提供结果。另外, 以防充分性造成影响, 我们选择看起来有变异充分的测试用例集的项目, 然后分别计算可能的优势。

4.1 实验

我们的任务是寻找每个项目的 $U_{perfect}$ 。对最好策略的要求很简单:

1. 变异体在整个变异体集中应该具有代表性。

即,

$$kill(T_p, M) = kill(T, M)$$

2. 由此所选择的变异体应该是非冗余的。

即,

$$\forall_{m \in M_p} kill(T_p, M_p \setminus \{m\}) \subset kill(T_p, M_p)$$

由 Ammann 等人[3]提出的最小变异体集满足我们对最好策略的要求, 因为它是有代表性的——一个测试用例集, 能够杀死最小变异体, 就可以杀死整个的变异体集——而且对与之对应的最小测试用例集, 它是非冗余的。

Ammann 等人[3]观察到, 最小变异体集的基数与对应的最小测试用例集的基数相同。即,

$$|M_{perfect}^{min}| = |MinTest(T, M)| = |T_{all}^{min}|$$

寻找变异体集的真正最小测试用例集是 NP 完全问题。最好的近似算法是 Chvatal[12]的, 使用贪心算法, 每次迭代都尽力选择能够最大数量覆盖变异体的集合。这在算法 1 中给出。最坏的情况下, 如果变异体的数量是 n , 可以覆盖它的最小测试用例是 k , 这个算法的近似度为 $k \cdot \ln(n)$ 。我们注意到, 该算法在实际中具有很强的鲁棒性, 通常获得的结果接近真实最小数量 k (见图 3)。此外, Feige[2]表明, 只要 $NP \neq P$, 这是集合覆盖算法可达到的最接近的近似比。

Algorithm 1 Finding the minimal test suite

```

function MINTEST( $Tests, Mutants$ )
     $T \leftarrow Tests$ 
     $M \leftarrow kill(T, Mutants)$ 
     $T_{min} \leftarrow \emptyset$ 
    while  $T \neq \emptyset \vee M \neq \emptyset$  do
         $t \leftarrow random(\max_t |kill(\{t\}, M)|)$ 
         $T \leftarrow T \setminus \{t\}$ 
         $M \leftarrow kill(T, Mutants)$ 
         $T_{min} \leftarrow T_{min} \cup \{t\}$ 
    end while
    return  $T_{min}$ 
end function
    
```

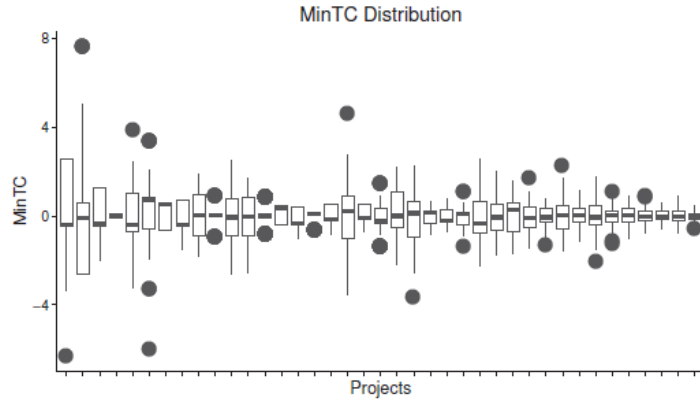


Figure 3: Variation of minimal test cases for each sample as a percentage difference from the mean ordered by mean minimal test suite size. There is very little variation, and the variation decreases with test suite size.

因为它是一种近似，所以我们贪婪估计的最小测试用例集的大小超过 100。变化在图 3 给出，按照最小测试用例集的大小顺序。注意，变化非常小，随着测试用例集大小的增加，变化减少。我们现在需要的就是，对于最好策略产生的相同数量的变异体，找到随机抽样的有效性。

接下来，我们从原始集合 M_{random} 中随机抽样 $|M_{perfect}^{min}|$ 个变异体，获得 T_{random}^{min} 样本的最小测试用例集，然后发现来自原始集合的变异体被这个测试用例集杀死 $kill(T_{random}^{min}, M)$ ，这个是用来计算相对于特定的随机抽样，最好策略的效用。对于每个项目，这个实验重复了 100 次，并取平均值来计算考虑范围内项目的 $U_{perfect}$ 。

5. 结果

5.1 所有变异体

我们的结果在 Table 3 中给出。我们发现，对 faunus 项目，最好策略取得的最大效用是 17.997%，然而对 joda-money 项目，最低效用是 1.153%。最好策略的平均效用是 13.078%。一个样本 u-test 表明，95% 的项目最大效用介于 12.218% 和 14.26% 之间 ($P < 0.001$)。图 6 是抓取的每个项目的效用分布。

用平均最小测试用例集的大小对项目进行分类。

Table 3: The maximum utility achievable by a perfect strategy for each project

| Project | $ kill(T, M) $ | $ kill(T_r, M) $ | U_{perf} |
|--------------------|----------------|------------------|------------|
| events | 702 | 662.97 | 0.06 |
| annotation-cli | 589 | 529.51 | 0.11 |
| mercurial-plugin | 401 | 342.91 | 0.17 |
| fongo | 1209 | 1052.99 | 0.15 |
| config-magic | 721 | 640.91 | 0.13 |
| claz | 1583 | 1402.39 | 0.13 |
| ognl | 12308 | 11426.09 | 0.08 |
| java-api-wrapper | 1304 | 1148.52 | 0.14 |
| webbit | 1981 | 1793.96 | 0.10 |
| mgwt | 1065 | 949.96 | 0.12 |
| csv | 1459 | 1282.93 | 0.14 |
| joda-money | 1272 | 1257.55 | 0.01 |
| mirror | 1440 | 1252.50 | 0.15 |
| jdbi | 4362 | 3914.73 | 0.11 |
| dbutils | 961 | 854.83 | 0.12 |
| cli | 2330 | 2069.84 | 0.13 |
| commons-math1-l10n | 2980 | 2527.66 | 0.18 |
| mp3agic | 4003 | 3620.41 | 0.11 |
| asterisk-java | 3206 | 2754.69 | 0.16 |
| pipes | 2176 | 1884.73 | 0.16 |
| hank | 7109 | 6200.08 | 0.15 |
| java-classmate | 2316 | 1969.76 | 0.18 |
| betwixt | 4271 | 3809.19 | 0.12 |
| cli2 | 3145 | 2760.66 | 0.14 |
| jopt-simple | 1718 | 1546.21 | 0.11 |
| faunus | 4809 | 4078.22 | 0.18 |
| beanutils2 | 1281 | 1141.73 | 0.12 |
| primitives | 4125 | 3565.83 | 0.16 |
| sandbox-primitives | 4125 | 3563.85 | 0.16 |
| validator | 4070 | 3616.71 | 0.13 |
| xstream | 9163 | 8307.12 | 0.10 |
| commons-codec | 8252 | 7455.50 | 0.11 |
| beanutils | 6823 | 6071.53 | 0.12 |
| configuration | 13766 | 12359.89 | 0.11 |
| collections | 8561 | 7392.63 | 0.16 |
| jfreechart | 32456 | 28171.19 | 0.15 |
| commons-lang3 | 26741 | 22742.46 | 0.18 |
| commons-math1 | 90681 | 81898.25 | 0.11 |
| jodatetime | 23796 | 20491.96 | 0.16 |

Table 4: The maximum utility achievable by a perfect strategy for each project using distinguishable mutants

| Project | $ kill(T, M) $ | $ kill(T_r, M) $ | U_{perf} |
|--------------------|----------------|------------------|------------|
| events | 59 | 49.15 | 0.20 |
| annotation-cli | 110 | 93.68 | 0.18 |
| mercurial-plugin | 102 | 80.95 | 0.26 |
| fongo | 175 | 145.13 | 0.21 |
| config-magic | 204 | 171.60 | 0.19 |
| claz | 151 | 129.24 | 0.17 |
| ognl | 2990 | 2835.77 | 0.05 |
| java-api-wrapper | 308 | 259.87 | 0.19 |
| webbit | 325 | 280.89 | 0.16 |
| mgwt | 168 | 140.60 | 0.20 |
| csv | 411 | 349.30 | 0.18 |
| joda-money | 236 | 230.76 | 0.02 |
| mirror | 532 | 444.17 | 0.20 |
| jdbi | 903 | 783.99 | 0.15 |
| dbutils | 207 | 170.60 | 0.21 |
| cli | 788 | 688.05 | 0.15 |
| commons-math1-l10n | 219 | 177.86 | 0.23 |
| mp3agic | 730 | 639.01 | 0.14 |
| asterisk-java | 451 | 372.25 | 0.21 |
| pipes | 338 | 288.41 | 0.17 |
| hank | 546 | 465.52 | 0.17 |
| java-classmate | 551 | 450.46 | 0.22 |
| betwixt | 1198 | 1055.30 | 0.14 |
| cli2 | 1066 | 903.30 | 0.18 |
| jopt-simple | 589 | 514.36 | 0.15 |
| faunus | 553 | 467.03 | 0.18 |
| beanutils2 | 465 | 392.30 | 0.19 |
| primitives | 1365 | 1155.09 | 0.18 |
| sandbox-primitives | 1365 | 1155.01 | 0.18 |
| validator | 759 | 647.36 | 0.17 |
| xstream | 1960 | 1691.84 | 0.16 |
| commons-codec | 1393 | 1192.29 | 0.17 |
| beanutils | 1570 | 1341.04 | 0.17 |
| configuration | 4522 | 3934.21 | 0.15 |
| collections | 2091 | 1750.05 | 0.19 |
| jfreechart | 4686 | 3910.15 | 0.20 |
| commons-lang3 | 4479 | 3663.98 | 0.22 |
| commons-math1 | 17424 | 15139.90 | 0.15 |
| jodatetime | 6920 | 5801.10 | 0.19 |

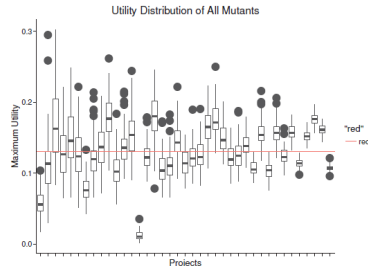


Figure 6: Using all mutants.

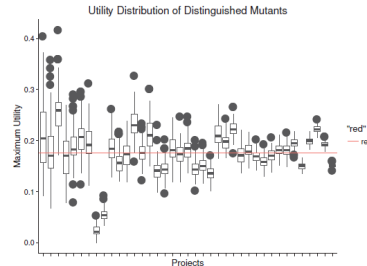


Figure 7: Using distinguished mutants.

Distribution of mean utility using distinguished mutants across projects. The projects are ordered by the cardinality of mean minimal test suite. The red line indicates the mean of all observations.

大家可能想知道这种情况是否会随着测试用例集的大小或者项目的大小得到改进。我们注意到，效用 U_p 与变异体总数、被检测的变异体（如图 5 所示）、变异得分和最小测试用例集（如图 4 所示）有较低的相关性。Table 5 中给出了相关因子。

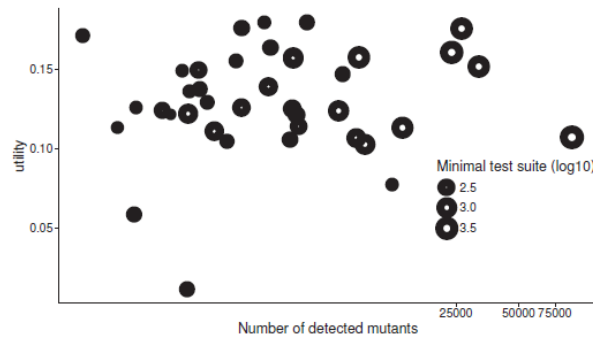


Figure 5: The figure plots utility (y-axis) against the number of detected mutants. Bubble size represents the magnitude of average minimal test suite size (\log_{10}). The figure suggests that there is no correlation between utility and number of detected mutants.

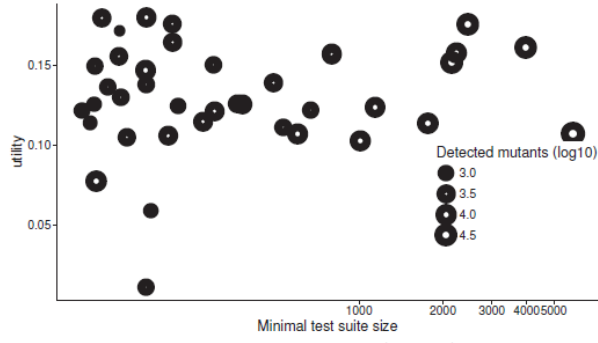


Figure 4: The figure plots utility (y-axis) against the average minimal test suite size (log10). Bubble size represents the magnitude of detected mutants (log10). The figure suggests that there is no correlation between utility and average minimal test suite size.

Table 5: The correlation of utility for all mutants, killed mutants, mutation score, and minimal test suite size, based on both full set of mutants, and also considering only distinguished mutants

| | R_{all}^2 | $R_{distinguished}^2$ |
|--------------|-------------|-----------------------|
| M | -0.02 | -0.03 |
| M_{kill} | -0.03 | -0.01 |
| M_{kill}/M | -0.02 | -0.00 |
| T^{min} | -0.01 | -0.02 |

确定显著影响 $U_{perfect}$ 的变量的方差分析 (ANOVA) 表明, 项目变化是一个重要因素 ($P < 0.001$), 而且和 $kill(T_{random}, M)$ 联系紧密。

$$\mu\{U_p\} = project + kill(T_r, M) + project \times kill(T_r, M)$$

变量项目和 $U_{perfect}$ 有 0.682 的相关性, 组合条件和 $U_{perfect}$ 有 0.9995 的相关性。

5.2 可区分变异体

我们的研究结果如 Table 4 所示。我们发现, 对项目 mercurial-plugin, 最好策略获得的最大效用是 26.159%, 然而 joda-money 项目的最低效用是 2.283%。

最好策略的平均效用是 17.545%。一个样本 u-test 表明, 95% 的项目最大效用介于 16.912% 和 18.876% 之间 ($P < 0.001$)。

图 7 是抓取的每个项目的效用分布。用平均最小测试用例集的大小对项目进行分类。

这种情况不会随着测试用例集的大小或者项目的大小发生改变。

效用 U_p 与变异体总数、被检测的变异体、变异得分和最小测试用例集有较低的相关性。Table 5 中给出了相关因子。

确定显著影响 $U_{perfect}$ 的变量的方差分析 (ANOVA) 发现, 项目变化是一个重要因素 ($P < 0.001$), 而且和 $kill(T_{random}, M)$ 联系紧密。

$$\mu\{U_p\} = project + kill(T_r, M) + project \times kill(T_r, M)$$

变量项目和 $U_{perfect}$ 有 0.734 的相关性, 组合条件和 $U_{perfect}$ 有 0.9994 的相关性。

5.3 充分的变异体

最后, 大家可能会问, 是否充分性对选择策略的有效性有影响。继工业实践认为在不计入等价变异体[47,52-54]后测试良好的项目是充分的, 我们选择大型的测试良好的且至少有 1000 个变异体, 变异得分至少为 70% (在上述相似研究的范围内) 的项目, 这些项目被认为是充分的。我们评估了 configuration、commons-lang3、commons-math1、jodatime 四个项目的效用, 发现它们的平均最大效用为 13.955%。这些相同的项目有一个突出的平均最大效用 17.893%。这表明充分性对选择策略的有效性没有明显的影响。

6. 讨论

变异分析是一个非常有价值的工具, 但是由于其较高的计算要求, 在实践中通常很难使用。有持续的和积极的研究, 通过使用不同的变异约简策略去补救这种情况。因此, 知道在最简单的约简策略——完全随机抽样的基础上, 可以改进的量是重要的。

我们对一个简单的理想化系统进行理论分析发现, 给定均匀分布的变异体, 具有变异体杀死预期知识的变异约简策略, 相对于随机抽样的平均改进比例为 58.2%。这个充当任何已知的变异约简策略可预期取得的上界 (假设变异体分布相当接近均匀)。

我们使用大量的开源项目进行的实证研究表明实际限制低得多, 然而, 对 PIT 产生的变异体, 平均仅为 13.078%。即使我们忽略偏差的影响, 仅仅使用可区分变异体, 可能的改进量平均为 17.545%。

区分理论和实证分析解决的不同问题是非常重要的。理论限制说明了在现有的变异体最差分布的情况下, 最好变异策略可达到的最好程度。另一方面, 实证分析发现, 在忽略不同程序的变异体分布的情况下, 最好策略的平均效用。然而, 考虑到偏差的影响相当弱 (仅为 4.467%), 因此理论边界对于实际问题同样是合理的。

实证得到的效用的上界出人意料的低, 并且对改进变异约简策略投入的精力存在质疑。当然, 大家仍然可以指出随机抽样是变幻莫测的, 一个人可以得到任意的好或坏的样本。然而, 我们的结果表明, 个别样本的方差相当

低，较大的项目的这种情况会得到很大的改善——例如，commons-math1 项目的方差仅为 0.397%。因此，在项目足够大到真的需要变异约简时，真的坏样本的可能性非常低，并且随着测试用例数量的增加急速下降。大家可能想知道测试用例的充分性是否有影响，但是，我们对具有充分测试用例的项目的分析表明，几乎没有不同（ $U_{\text{perfect}}=13.955\%$ ）。在一般情况下，使用公认的标准规范进行统计抽样，产生合理大小的随机变异体样本应该是切实有效的，以避免由于随机不确定性造成的不常见的不良结果。增加的优点是，随机抽样是很容易实现的，并导致可忽略不计的开销。

我们注意到，我们的框架不仅适用于选择性变异，而且也适用于希望添加新的应用程序的变异用户。如果一个变异实现者拥有一个完备的变异算子集，那么他们当前的变异体集没有任何冗余变异体（考虑到我们对变异符号学的肤浅理解，实际是几乎不可行的）。即使我们认为增加一个新的随机变异体集对改进变异集没有任何效果，而且相对于原始集合而言，它们是冗余的（在实践中很少见，因为我们正在引进新的变异体），因此我们的限制造成的最大缺点是边界化（对 95% 的项目上限为 18.876%）。然而，和可能的错误相比，至少一些新的变异体可以被认为能够提高变异体集的代表性。因为我们不能约束可被引入的可区分变异体的数目，添加新的变异算子带来的最大优势不存在上限。最新的结果表明，真正故障的种类没有耦合到任何目前常用的变异算子[32]中，根据这些结果，添加新的变异算子是极其具有吸引力的。

我们以前的研究[25]表明，恒定数量的变异体（理论最大值为 9604 和 1000，在实践中 1% 的精度）是足够的，对于在不考虑总的变异体数量的情况下计算高准确度的变异得分。这表明，采样既不会孙氏有效性，也不会损失精度，因此增加新的变异算子（和采样所需数量的变异体）是潜在的一个非常有成效的方式。

7. 有效性威胁

虽然我们已经注意确保我们的结果是公正的，并试图消除随机噪声的影响。随机噪声可导致选择的项目、工具或语言不具有代表性，并可能导致实证结果存在偏差的层和误差。我们的研究结果受到以下威胁。

近似带来的威胁：我们使用 Chvatal[12]的贪心算法近似最小测试用例集的大小。虽然这是保证是 $H(|M|)$ 近似，但仍有一定的误差范围。我们为每一个观察点平均运行 100 次来防止这个错误。其次，我们采用随机抽样方法对随机抽样的效果进行评价。虽然我们每一个观察每次使用了 100 个试验，偏差还是可能存在。

抽样偏差带来的威胁：为保证样本的代表性，我们选择使用来自 Maven

构建系统构建的 GitHub 库的 java 项目。对给定的 GitHub API，我们挑选所有可以检索的项目，并仅基于构建和测试条件，从这些项目中进行选择。然而，我们的样本程序在 Github 上获取的项目中可能存在偏差。

使用的工具造成的偏差：我们的研究依赖于 PIT。我们已经尽我们最大的努力去扩展 PIT，为的是提供一个合理充分的变异算子集，也确保变异算子是非冗余的。此外，我们已经尝试通过考虑可区分变异体的效果，来减少冗余的影响。还有一种可能性就是产生的变异体类别可能有偏差，这可能会影响我们的分析。因此，未来这项研究需要用多种工具产生的变异体和项目进行复现。

8.结论

我们的研究表明，盲随机抽样的变异体是非常有效的，和使用变异分析结果完美知识的变异约简策略可获得的最佳可实现的界限相比，令人惊讶的是改进空间很小。之前的研究表明，和随机抽样相比，目前的变异算子选择策略几乎没有优[53,54]。然而，实验缺少与随机抽样相同数量变异体的直接比较。其次也表明，当和真实最小变异体集相比时，目前的策略相当差，但是缺乏和随机抽样的比较。我们的贡献表明，忽略策略的智能性，任何约简策略的改进都有一个理论上的限制，也有一个最好策略和随机抽样有效性的直接实证比较。

我们的理论研究表明，对于给定的任意程序，假设在冗余变异体中没有偏差，具有变异体杀死预期知识的最好变异约简策略，相对于随机抽样的平均优势为 58.2%。实证方面，我们发现具有预期知识的最好约简策略有一个非常低的优势 13.078%。

平均仅为 13.078%。即使我们忽略冗余变异体群中偏差的影响，仅仅使用可区分变异体，我们发现和随机抽样相比，最好变异约简策略的优势仅为 17.545%。偏差（4.467%）的低影响表明，我们理论分析的简化假设并没有与得分偏离很远。理论预测和实证结果之间的差距是由现实世界测试用例集的不充分性导致的，造成和可区分变异体集相比，最小变异体集小的多。我们注意到，变异约简策略经常声称为高约简因素，而且大家可能会认为和随机抽样相比效用相似，这无论是在理论或实践中都未实现。

我们的研究的第二个收获是，对于给定的限制效用，研究人员或变异测试工具的执行者应仔细考虑实施一个变异约简策略的价值。事实上，我们的研究[24]表明，我们检测的流行的变异算子选择策略，和随机抽样相比，效用降低了，甚至基于程序元素的分层取样技术很少超过 10% 的改善。鉴于项目的变异性是重要的，一个测试从业者也会很好地考虑是否使用的变异约简

策略适用于测试的特定的系统（也许根据这个项目的历史数据，或者一些既定的意义类似的项目）。随机抽样的变异体不是非常远离一个理想的变异约简策略的实证上限，并且，由于一个可能无法为一个给定的项目很好工作的“聪明”的选择方法，对于一个意料之外的偏见几乎没有空间。这里报告的限制是基于使用变异杀死矩阵的完全知识，至少，这在实践中是难以获得的。

也许我们的研究能拿出的最重要结论是，不是通过移除变异算子，而是通过进一步研究新的变异算子（或新的变异算子类别，例如针对并发和资源请求的特定域的变异算子），提高变异分析的有效性是可能的。我们的研究表明，尽管可实现的改进程度没有限制，但通过增加新的变异算子取得的最大约简效用仅为 23.268%。

在学取得成果

一、 在学期间所获的奖励

| 奖励名称 | 授奖机构 | 授奖时间 |
|-----------------------------------|---|---------|
| 优秀三好学生 | 北京科技大学 | 2013.11 |
| 北京市物理竞赛三等奖 | 北京物理学会 | 2013.12 |
| 十佳志愿者 | 北京科技大学 | 2014.06 |
| 优秀学生干部 | 北京科技大学 | 2014.11 |
| 国家奖学金 | 中华人民共和国教育部 | 2014.11 |
| 第十六届“摇篮杯”大学生课外学术科技作品竞赛三等奖 | 北京科技大学 | 2014.12 |
| 第九届国际大学生 iCAN 创新创业大赛 2015 年总决赛三等奖 | 中国微米纳米技术学会/全球华人微纳米分子系统学会/国际大学生 iCAN 创新创业大赛组委会 | 2015.10 |

二、 在学期间发表的论文

Yuhan Liu, Luxi Xing, Xiang Tian, Zhen Wang. Application of Monte Carlo & Optimized UCT Algorithm in WTN Chess [C]. Chinese Control and Decision Conference. 2016: 3422-3427.

三、 在学期间取得的科技成果

致 谢

首先感谢孙昌爱老师对本次毕业设计的耐心指导与大力支持。本次毕业设计从选题、开题、中期以及最终毕业论文的完成都离不开孙老师的帮助与指导。在毕设期间，孙老师严谨的工作态度，求实的治学精神和渊博的专业知识给我留下了深刻的印象。在此，谨向孙老师表示崇高的敬意和衷心的感谢！

同时还要感谢实验室的师兄师姐们，感谢他们在百忙之中抽出时间悉心为我答疑解惑。感谢潘琳师姐对我的指导和鼓励，她在我的毕设完成过程中提供了很大的帮助。无论是在理论知识、实验设计、工具设计还是各个报告的撰写方面，潘琳师姐都给予了我专业的指导。正是由于他们的帮助，我才能解决问题，顺利完成毕业设计论文。

感谢大学四年里帮助我的老师及同学，同时也感谢我的父母，感谢他们让我拥有一个充实而美好的大学生活。在此，我谨向所有鼓励、帮助我的人致以诚挚的谢意！