# 摘　　要

软件测试的根本问题是如何通过较少的测试用例尽可能多的检测出程序中隐藏的缺陷。作为一种基于故障的软件测试技术，变异测试可用于评估测试用例集的测试充分性，也可以通过模拟被测软件的真实缺陷，从而对已有测试方法的有效性进行辅助评估。

然而测试过程中带来的大量计算开销阻碍了该技术在工业界的应用。很多学者致力于变异测试技术的优化与改进，以期望找到合适的方法来减小计算开销。

本文从变异测试执行开销角度出发，探索在不影响故障检测能力的前提下提高变异测试效率的方法，提出了一种基于并发机制的变异测试优化技术，通过实验评估验证该技术的有效性。设计并实现了相关支持工具，用于验证所提方法的实用性。本文取得的主要研究成果如下：

* **提出了一种基于并发机制的变异测试优化技术：**将变异测试与程序分析相结合，分析变异体集合在程序块的分布情况；针对每个程序块，将相关的变异体集合与原始程序合二为一，变异代码处理为并发分支；执行合成后的程序，依据原始程序与变异体的输出结果判定相应变异体是否被杀死，如果杀死则新的测试用例无需再执行该变异体所在并发分支，否则依旧执行该分支，直到所有变异体均被检测出来或所有测试用例均被执行完毕，此次测试结束。
* **开发了基于并发机制的变异测试优化技术支持工具：**设计与实现基于并发机制的变异测试优化技术支持工具，实现了变异体分组、程序合成、测试用例执行、测试结果验证等功能。
* **采用实例研究评估了基于并发机制的变异测试优化技术的有效性与可行性：**采用了2个程序，比较传统变异测试方法与优化方法下变异体执行所需时间，验证了基于并发机制的变异测试优化技术的有效性及支持工具的实用性。

基于并发机制的变异测试优化技术减少了变异测试过程中变异体编译和执行的计算开销，开发的测试工具提高了该技术的自动化程度。利用本文开发的并发变异测试技术与工具，测试人员能够较为快速的进行变异测试。

**关键词：变异测试，并发机制，Java图形用户界面，测试工具**

**Research on Concurrent Mechanism Based Mutation Testing Technique and Its Supporting Tool**

# Abstract

The basic problem of software testing is how to use the least test cases to test more defects hidden in programs. As a fault-based testing technique, mutation testing can be used to evaluate the testing sufficiency of test suites. It can also assist in estimating the effectiveness of the existing testing method by simulating the real faults of the software which is tested.

However, the abundant computing cost produced during the testing process hinder the application of mutation testing in industrial community. Many scholars devote themselves to optimize and improve this technology, looking forward to find a suitable way to reduce computing cost.

From the perspective of computing cost produced by executing mutation testing, this thesis explores the method of improving the efficiency of mutation testing under the premise of no effect on the capability of fault detection. Basing on concurrent mechanism, an improved mutation testing technology is brought up and experiments are used to estimate the validity of this technique. A tool is designed and actualized to support the improved mutation testing so that the practicability of the brought up method can be tested. The main achievements of this paper are as follows:

* **An concurrent mechanism based mutation testing:** Under the combination of mutation testing and program analysis, programs are divided into basic blocks and the distribution of mutation sets is analyzed; as to every program block, the mutant sets related to the block are combined with the original program into a new program, the mutation codes are handled for concurrent branches; executing the combined program, depending on the output of the mutants and the original program to determine whether the corresponding mutants are killed, if a mutants is killed then the new test case do not need to execute the concurrent branch which contains the died mutant, otherwise the concurrent branches which do not contain any died mutants are still carrying out until all the mutants are found out or all the test cases have been tested.
* **A tool to support** **the concurrent mechanism based mutation testing method:** A tool is designed and actualized to support the improved mutation testing method, which includes functions such as dividing mutants into groups, composing programs, executing test cases, proving test results and so on.
* **Effectiveness evaluation of the concurrent mechanism based mutation testing method and tool:** Two example programs are used to compare the executing time between tradition mutation testing method and the improved mutation testing technology basing on concurrent mechanism, proving the effectiveness of the improved mutation testing technology and the practicability of the supporting tool.

The mutation testing technology basing on concurrent mechanism cuts down the computing cost and the compiling time of mutants during the process of mutation testing. The developed testing tool has improved the automation level of this technology. By using the concurrent mutation testing tool developed in this thesis, testers can finish the mutation testing faster.

**Key Words：Mutation Testing; Concurrent Mechanism; Java Graphical User Interface; Testing Tool**

**目　　录**

[摘　　要 1](#_Toc421660203)

[Abstract 2](#_Toc421660204)

[插图或附表清单 4](#_Toc421660205)

[引　　言 1](#_Toc421660206)

[1文献综述 2](#_Toc421660207)

[1.1国内外研究现状 2](#_Toc421660208)

[1.1.1国内研究进展 2](#_Toc421660209)

[1.1.2国外研究进展 2](#_Toc421660210)

[1.2课题来源 4](#_Toc421660211)

[1.3研究内容及成果 4](#_Toc421660212)

[1.4论文组织结构 5](#_Toc421660213)

[2背景介绍 6](#_Toc421660214)

[2.1变异测试 6](#_Toc421660215)

[2.1.1变异测试技术概述 6](#_Toc421660216)

[2.1.2变异测试技术优缺点 8](#_Toc421660217)

[2.2 Linux进程与线程 9](#_Toc421660218)

[2.3 Linux操作系统创建进程的方法 9](#_Toc421660219)

[2.3.1 fork进程创建方法 9](#_Toc421660220)

[2.3.2 vfork进程创建方法 11](#_Toc421660221)

[2.3.3 clone进程创建方法 11](#_Toc421660222)

[2.4 Eclipse平台及Java Swing用户界面开发技术 12](#_Toc421660223)

[2.5小结 13](#_Toc421660224)

[3基于并发机制的变异测试优化技术 14](#_Toc421660225)

[3.1基本思想 14](#_Toc421660226)

[3.2方法简述 16](#_Toc421660227)

[3.3具体实现 24](#_Toc421660228)

[3.4小结 30](#_Toc421660229)

[4基于并发机制的变异测试优化支持工具的设计与实现 31](#_Toc421660230)

[4.1需求分析 31](#_Toc421660231)

[4.2架构设计 34](#_Toc421660232)

[4.3工具的实现 35](#_Toc421660233)

[4.4工具演示 37](#_Toc421660234)

[4.5小结 41](#_Toc421660235)

[5实验评估 43](#_Toc421660236)

[5.1研究问题 43](#_Toc421660237)

[5.2度量指标 43](#_Toc421660238)

[5.3实验对象 43](#_Toc421660239)

[5.3.1 minmax求最大值最小值程序 43](#_Toc421660240)

[5.3.2 nextdate求日期程序 44](#_Toc421660241)

[5.4实验过程 45](#_Toc421660242)

[5.5实验结果分析 48](#_Toc421660243)

[5.5.1基于并发机制的变异测试优化技术与传统方法的测试执行时间对比 48](#_Toc421660244)

[5.5.2基于并发机制的变异测试优化技术的可行性与适用性 57](#_Toc421660245)

[5.5.3基于并发机制的变异测试优化技术对变异得分的影响： 57](#_Toc421660246)

[5.6小结 58](#_Toc421660247)

[结　　论 59](#_Toc421660248)

[参 考 文 献 60](#_Toc421660249)

[附 录 63](#_Toc421660250)

[附录A 外文文献 63](#_Toc421660251)

[附录B 外文翻译 81](#_Toc421660252)

[附录C minmax程序源代码 94](#_Toc421660253)

[附录D nextdate源代码 95](#_Toc421660254)

[附录E 实验原始数据 97](#_Toc421660255)

[在 学 取 得 成 果 99](#_Toc421660256)

[致　　谢 100](#_Toc421660257)

# 插图或附表清单

[图2.1 传统变异测试分析流程 7](#_Toc421473908)

[图2.2 fork函数执行流程图 10](#_Toc421473909)

[图2.3 Eclipse平台体系结构图 12](#_Toc421473910)

[图3.1 minmax原始程序及变异体部分代码 14](#_Toc421656374)

[图3.2 传统变异测试方法针对程序块Bi的变异体执行过程 15](#_Toc421656375)

[图3.3 基于并发机制的变异测试优化方法针对程序块Bi的变异体执行过程 16](#_Toc421656376)

[图3.4 传统变异测试流程 17](#_Toc421656377)

[图3.5 基于并发机制的变异测试优化技术流程 18](#_Toc421656378)

[图3.6 多进程创建流程图 20](#_Toc421656379)

[图3.7 多进程创建方法处理后的示例程序代码 22](#_Toc421656380)

[图3.8 无效测试用例示例代码 24](#_Toc421656381)

[图3.9 基于并发机制的变异测试优化技术的实现过程 24](#_Toc421656382)

[图4.1 CMC系统用例图 31](#_Toc421660184)

[图4.2 工具目录结构 33](#_Toc421660185)

[图4.3 CMC工具架构 34](#_Toc421660186)

[图4.4 待测程序选择界面 38](#_Toc421660187)

[图4.5 变异体分组界面 39](#_Toc421660188)

[图4.6 导入分组规则文件以及变异体集合后的反馈信息 39](#_Toc421660189)

[图4.7 变异体合并界面 40](#_Toc421660190)

[图4.8 测试执行界面 41](#_Toc421660191)

[图4.9 测试报告内容及格式 41](#_Toc421660192)

[图5.1 minmax程序全部变异体执行时间对比 45](#_Toc421535226)

[图5.2 minmax程序全部变异体执行平均时间对比 45](#_Toc421535227)

[图5.3 minmax程序Block1相关变异体执行时间对比 46](#_Toc421535228)

[图5.4 minmax程序Block2相关变异体执行时间对比 46](#_Toc421535229)

[图5.5 minmax程序Block3相关变异体执行时间对比 46](#_Toc421535230)

[图5.6 minmax 程序Block4相关变异体执行时间对比 47](#_Toc421535231)

[图5.7 minmax程序变异体分组执行平均时间对比 47](#_Toc421535232)

[图5.8 所属不同程序块的minmax程序变异体执行时间缩减比例 48](#_Toc421535233)

[图5.9 nextdate程序全部变异体执行时间对比 48](#_Toc421535234)

[图5.10 nextdate程序全部变异体执行平均时间对比 49](#_Toc421535235)

[图5.11 nextdate程序Block1相关变异体执行时间对比 49](#_Toc421535236)

[图5.12 nextdate程序Block2相关变异体执行时间对比 50](#_Toc421535237)

[图5.13 nextdate程序Block3相关变异体执行时间对比 50](#_Toc421535238)

[图5.14 nextdate程序Block4相关变异体执行时间对比 50](#_Toc421535239)

[图5.15 nextdate程序Block5相关变异体执行时间对比 51](#_Toc421535240)

[图5.16 nextdate程序Block6相关变异体执行时间对比 51](#_Toc421535241)

[图5.17 nextdate变异体分组执行平均时间对比 52](#_Toc421535242)

[图5.18 所属不同程序块的nextdate程序变异体执行时间缩减比例 53](#_Toc421535243)

[表2.1 典型等价变异体示例 6](#_Toc421474028)

[表2.2 典型变异算子示例 7](#_Toc421474029)

[表2.3 三种进程创建方法对比 12](#_Toc421474030)

[表3.1 示例程序及变异体代码 21](#_Toc421474038)

[表4.1 工具目录名称及其功能 32](#_Toc421535267)

[表5.1 minmax程序变异体个数统计 40](#_Toc421535313)

[表5.2 minmax程序变异体分组情况统计 40](#_Toc421535314)

[表5.3 nextdate程序变异体个数统计 41](#_Toc421535315)

[表5.4 nextdate程序变异体分组情况统计 41](#_Toc421535316)

[表5.5 传统变异测试方法实验目录 41](#_Toc421535317)

[表5.6 基于并发机制的变异测试优化方法实验目录 42](#_Toc421535318)

# 引　　言

早期，软件测试被错误地定位为“试图证明程序是正确的”，然而Dijkstra在1972年提出“软件测试只能说明软件中存在错误，不能表明软件中不存在错误”[1]。软件测试的目的是为了进一步提高软件的质量，尽可能地检测并消除软件中潜藏的各种故障。经过多年软件测试的实践和测试理论的研究工作，人们发现，要想找出程序中潜藏的所有错误几乎是不可能的，因此软件测试总是在测试成本与程序中未发现故障的潜在成本之间进行权衡。

为了做出明智的测试决定，需要提供一种方法来评估测试工作的质量（给定一个测试用例集，预测其是否具有有效的故障检测能力），作为测试是否终止的标准。在软件测试的研究中，测试用例的评价标准通常被认为是实际故障检测能力。目前，变异测试是一种比较高效的测试手段。

变异测试作为一种基于故障的软件测试技术[2]，可用于衡量测试数据集的发现错误的能力，评估和改进测试用例集的测试充分性，也可以通过采用变异缺陷来模拟被测软件的真实缺陷，从而对研究人员提出的测试方法的有效性进行辅助评估[3]。然而测试过程中带来的大量计算开销阻碍了变异测试在工业界的应用，其开销主要分布于变异体的生成、编译和执行过程中[3]。为了将变异测试技术从学术界研究转化到工业界应用，很多学者致力于变异测试技术的优化与改进，以期望找到合适的方法来减小计算开销。

本课题从变异测试执行开销角度出发，将程序分析与并发机制引入变异测试过程中，通过并发机制对传统的变异测试技术进行优化，探索在不影响故障检测能力的前提下提高变异测试效率的方法，提出了一种基于并发机制的变异测试优化技术，能够有效降低变异测试执行时间开销；采用Java Swing图形界面开发技术设计与实现基于并发机制的变异测试优化技术支持工具，方便验证所提方法的有效性及应用性。所研究的相应技术与工具有望实现减少变异体执行过程中的时间开销，促进变异测试技术在工业界应用。

# 1文献综述

## 1.1国内外研究现状

### 1.1.1国内研究进展

清华大学的姜凡和郑人杰[4]提出了一种通过弱变异测试结果自动补足测试数据的方法，并基于关系反馈概念提出了增大测试覆盖面的一种关系测试数据生成算法。

北京大学的单锦辉提出了一种自动生成检测多个在同一位置的变异体的测试数据的方法，并结合面向路径的测试数据自动生成方法，以提高生成测试数据集的效率和有效性[5]。

上海交通大学的陆毅明提出了一种基于代数式规范的变异测试方法[6]，与传统变异测试方法（基于代码）相比，这种变异方法能够检测出由于错误理解需求规范而引入的故障，生成更少的变异体，从而提升变异测试效率。

针对变异测试中变异体数量缩减技术，张路等人提出一种在随机选择变异算子的基础上再次随机选择变异体的双轮选择方法[7]，徐拾义则通过故障控制和等价关系来减少需要注入的故障数[8]。

本课题组针对软件故障分布的“群束”特征，提出了一种非均匀分布的变异分析技术，并应用该技术对Mujava工具进行了扩展与改进，该技术增强了传统变异分析技术对测试技术的性能评估的可信度，在减少变体数量的基础上选出具有群束特征的变体集合，降低了变异测试的开销[9]。同时研究了将变异测试应用于BPEL程序中的关键问题，提出了面向BPEL程序的变异测试框架，开发相应支持工具MuBPEL。

### 1.1.2国外研究进展

近年来，研究人员尝试从多个方面改进与拓展变异测试，包括正确识别等价变异体以及如何减少变异分析的开销。已有的研究工作侧重于降低变异测试的开销，主要分为减少变异体数量和降低变异体执行时间[10]。

在精简变异体数量方面，主要研究进展如下：

1. 随机选择法：随机选择法尝试按照一定比例x，从执行变异算子生成的变异体集合M中随机选出M•x%的变异体，测试时仅执行这个变异体子集。Mathur和Wong[10]为找到合适的选择比例x，针对Mothra系统[18]中的22种变异算子展开了研究，结果表明，当选择比例为10%时，构造的测试用例集的测试充分性仅比用所有变异体构造的测试用例集减少16%，并且选择比例越高，效果越好。
2. 变异算子选择法：相对于随机选择法，变异算子选择法则期望在不影响变异得分的前提下，减少变异算子的使用个数来缩减生成的变异体个数。Mathur在变异测试和分析中发现，不同类型的变异算子生成的变异体数量差别较大，并提出“约束变异”概念[11]，即选择一部分变异算子，生成较少变异体的方法；由于变异算子仅对原始程序做合乎语法的微小改动，因此某些变异算子会产生大量变异体，Offutt等人在此基础上建议在进行变异算子选择时忽略生成变异体较多的变异算子[12]。通过对Mothra系统的22种变异算子进行实验后，指出采用算数运算符号替换(AOR)算子、关系运算符号替换(ROR)算子、逻辑连接符替换(LCR)算子、一元运算符插入(UOI)算子和绝对值插入(ABS)算子生成的变异体数量减少77%，其测试用例集的变异分数可达到采用全部变异算子变异得分的99% [7]。
3. 高阶变异测试：Jia和Harman提出了高阶变异体概念[13]，即在原有程序P上依次执行多次变异算子形成变异体P’[3]。一个高阶变异体是由多个单阶变异体构成，若采用高阶变异体代替单阶变异体，可以有效缩减变异体数量。Polo等人提出3种将一阶变异体组合成二阶变异体的构造算法，将两个一阶变异体组合成一个二阶变异体，可以缩减一半变异体集合数量，实验结果表明：采用二阶变异体不会显著降低测试的有效性，但却可以有效减少测试开销[14]。

在降低变异体执行时间方面，主要研究进展如下：

1. 变异体检测优化：对于变异体检测优化的研究，国外提出了多种方法，大多方法是应用弱变异[15]或固定变异[3]检测技术来提高变异体检测效率。强变异检测最早由DeMillo 等人提出[2]，当一个测试用例执行原始程序与变异体之后的输出结果不同时，该变异体被“杀死”，需要满足可达性（变异体变异代码部分必须能够达到）、必要性（一旦到达，必然引起错误状态）和充分性（错误状态可以传播到输出结果）。弱变异仅需满足前两个性质，即认为变异体被杀死，采用弱变异检测技术时，仅需执行完变异体故障部分，无需执行完所有程序即可判断该变异体是否被杀死，从而提高了变异测试检测效率。
2. 变异体编译优化：King和Offutt首先将原有程序转化为中间代码，然后对该中间代码进行变异和解释执行，对变异体编译过程提出了优化方法[3]；随后许多学者开展了针对编译优化技术的研究工作。
3. 并行执行变异体：除了上述优化技术，研究人员也提出借助计算机体系结构，采用并行执行方式来进一步提高变异测试分析效率。如借助单指令多数据计算机、多指令多数据计算机提高变异测试分析效率[16]。

## 1.2课题来源

中央高校基本科研业务费资助课题“高效新型的SOA软件测试技术与工具研究”

## 1.3研究内容及成果

已有的基于并行计算机的并行优化方法依赖于计算机硬件基础，本课题探索基于软件并发机制的变异体执行方法，利用并发机制对传统的变异测试技术进行优化，提出基于并发机制的变异测试优化方法，期望可以减少变异体执行时间，提高变异测试的效率。

将程序分析与变异测试相结合，对变异体集合进行分析，将其分组并将同组变异体集合合并为支持并发运行机制的统一程序。通过实验评估的方式，对比本课题提出的优化方法与传统变异测试方法对相同数量变异体的执行时间，验证所提优化方式的有效性。最后，在Eclipse平台下实现基于并发执行的变异测试优化支持工具。

涉及相关知识如下：

1. 了解Linux下进程结构及多进程编程方法，掌握fork函数基本框架；
2. 掌握shell脚本编程方法，实现测试用例的执行、分析执行结果、统计实验时间等功能；
3. 掌握Java Swing基础组件的使用方法，学习Java Swing用户界面开发。

主要解决如下关键问题：

1. 变异体集合分布分析：对程序进行基本块划分，分析相应程序块上的变异体集合；
2. 基于程序块的变异体合成技术：针对某个程序块将原始程序与该块相关变异体集合合成一个程序，变异代码需处理为并发分支；
3. 基于程序块的变异体并行执行方法：执行原始程序至发生故障前的所有程序块，并发执行该程序块相关的所有变异体后续代码及原始程序后续代码。依据原始程序与变异体的输出结果判定相应变异体是否被杀死，如果杀死则新的测试用例无需再执行该变异体所在并发分支，否则依旧执行该分支，直到所有变异体均被检测出来或所有测试用例均被执行完毕。

预期目标如下：

1. 掌握Linux操作系统的基本操作，熟悉Linux下的C语言程序开发环境；利用GCC编译器、GDB调试命令及Shell脚本实现编译、执行、调试C语言程序，分析对比变异体与原始程序的执行结果等功能；
2. 通过实验评估的方式，对比基于并发机制的变异测试方法与传统变异测试方法的执行时间，验证所提优化方法的有效性；
3. 在Eclipse平台上设计与实现基于并发机制的变异测试工具，并对开发的测试工具进行验证。

## 1.4论文组织结构

本文的组织结构安排如下：

1. 介绍国内外研究现状、课题来源、研究内容及成果；
2. 介绍课题相关概念及技术。包括变异测试技术概述、Linux进程与线程、Linux操作系统下创建进程的方法、Eclipse平台及Java Swing用户界面开发技术；
3. 深入讨论基于并发机制的变异测试优化技术，主要包括对技术的基本思想、方法简述，具体实现；
4. 讨论基于并发机制的变异测试优化支持工具的设计与实现，包括需求分析、架构设计、工具实现中的关键问题以及系统演示；
5. 对基于并发机制的变异测试优化技术的可行性与有效性进行实验评估，主要包括研究问题阐述、实验设计与过程及实验结果分析。

# 2背景介绍

文章介绍本文研究的相关概念及技术，包括变异测试、Linux操作系统创建进程的方法、Eclipse平台及Java Swing用户界面开发技术。

## 2.1变异测试

### 2.1.1变异测试技术概述

变异测试(Mutation Testing)又称为变异分析，是一种典型的基于故障的测试技术[2]。其概念的提出最早可以追溯到Hamlet和DeMillo等人在20世纪70 年代的研究工作[10]。随后研究人员对该领域展开了深入的研究，推进了变异测试的应用。

传统变异测试一般通过生成与原始程序差异极小的错误程序来充分模拟被测软件可能存在的缺陷。其可行性基于“熟练程序员”和“耦合效应”假设[10]：通过对被测程序作合乎语法的代码修改，可以模拟熟练程序员的实际编程行为；若测试用例可以检测出简单缺陷，则该测试用例也易于检测出更为复杂的缺陷。

变异测试基本思想是：测试人员首先根据待测程序的特征向待测程序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[2] (Mutant)，用来模仿某种故障的操作称为“变异算子”(Mutation Operator)，变异算子一般在符合语法前提下仅对待测程序作微小改动。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)。若不存在任何测试用例，使在某一变异体与待测程序上的执行结果不同，则称该变异体为等价变异体[17](Equivalent Mutants)，如表2.1所示给出了一个典型的等价变异体示例。变异算子将被测程序P中循环条件“j<10”变异为“j!=10”，循环体中不存在修改变量j的语句，可以看出程序P’是程序P的等价变异体。

表2.1 典型等价变异体示例

|  |  |
| --- | --- |
| 程序P | 变异体P’ |
| for (int j = 0;j<10;j++)  sum += a[j]; | for (int j = 0;j!=10;j++)  sum += a[j]; |

变异运算是变异测试的关键，通常考虑算术运算符之间的替换、变量之间的替换、关系运算符之间的替换、变量与常量之间的替换以及逻辑运算符之间的替换。表2.2给出了5个典型的变异算子的例子。

表2.2 典型变异算子示例

|  |  |  |
| --- | --- | --- |
| 变异类型 | 程序P | 变异体P’ |
| 算术运算符之间的替换 | if (a + b >c)  return true; | if (a – b > c)  return true; |
| 变量之间的替换 | FUNCTION MAX (M,N)  MAX=M; | FUNCTION MAX (M,N)  MAX=N; |
| 关系运算符之间的替换 | IF (N>M)  MAX = N; | IF (N<M)  MAX = N; |
| 变量与常量之间的替换 | if (a + b >c)  c=a+b; | if (a + b >c)  c=a+1; |
| 逻辑运算符之间的替换 | if(a+b>c&&a+c>b)  flag=ture; | if(a+b>c||a+c>b)  flag=ture; |

传统的变异测试分析流程[3]如图2.1所示。



图2.1 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，传统变异测试步骤如下：

1. 根据待测程序P的特征设定一系列变异算子，通过在原有程序P上执行变异算子生成大量变异体。
2. 从所有变异体中识别出等价变异体。
3. 添加一个测试用例t至测试用例集T中。
4. 在非等价变异体上执行测试用例集T中的测试用例t，与原始程序的输出结果进行比较。
5. 如果与原始程序输出结果不同，则此变异体被杀死；否则，该变异体存活。
6. 若可以检测出所有非等价变异体则变异测试分析结束，否则对未检测出的变异体，需要额外设计新的测试用例，并添加到测试用例集T中，重复步骤3到5。

针对给定的测试用例集，能够“杀死”的变异体数量占所有产生的非等价变异体数量的百分比，称为该测试用例集合的“变异得分”[18](Mutation Score)。变异得分计算公式如下：

式(2-1)

式中：P表示被测程序；T代表测试数据集；D表示被杀死的变异体数量；M表示产生的变异体总数；E代表等价的变异体数量。变异得分可以用来评估测试用例集的缺陷检测能力，这也是变异测试的主要目的，最优目标是希望变异得分为100%，即测试用例集可以检测出所有的非等价变异体。

该技术除了可用于衡量测试用例集(Test Suite)的揭错能力[19]；也可以通过植入错误来系统地模拟被测软件的各种真实缺陷，对已有测试用例集进行辅助评估。变异测试不仅成功应用于C、C#、Java、SQL等不同的编程语言中，开发出了Mothra、Proteum、MuJava、Muclipse等测试支持工具，并成功应用于测试规格说明书和程序模型上[20]。主要应用于单元测试，当前在接口测试、面向对象软件的测试和合约测试等方面都有相关研究进展[21]。

### 2.1.2变异测试技术优缺点

在软件测试的理论研究中，测试用例的质量通常用测试用例杀死变异体的能力评估[22]。研究结果表明，变异分析的故障检测能力强，与手工植入缺陷相比，自动生成的变异体模拟真实软件中的故障效果更佳[9]，具有排错能力强、方便灵活、自动化程度高等优点[5]。

尽管如此，变异分析技术并没有广泛应用于工业界，主要原因如下：

1. 变异测试分析过程中的计算开销较大；
2. 等价变体难以检测（即不存在一个测试用例使得该变体与原始程序产生不同的输出）；
3. 缺少高效的自动化变异测试支持工具。

变异测试技术的缺陷极大程度地阻碍了变异测试在软件产业界的广泛使用。因此如何提高变异测试的效率成为了很多学者的研究重点。

## 2.2 Linux进程与线程

Linux是多任务操作系统，允许多个进程并发执行[23]，进程是系统执行程序和分配资源（CPU，内存，文件等）的基本单位，一般Linux下的进程包含以下几个关键要素[24]：一段可执行程序、专用的系统堆栈空间、私有的“PCB进程控制块”以及独立的存储空间。

在Linux操作系统中进程标识有进程号(PID)及其父进程号(PPID)。子进程的栈、数据以及栈段开始时是父进程内存相应各部分的完全拷贝，独立于父进程，具有良好的并发性。

线程是进程内的一个执行单元或可调度实体，共享进程拥有的代码、数据、文件等资源；有自己的堆栈、程序计数器等私有资源[23]。如果这时一个线程修改了某个全局变量，这将影响到其他线程。

## 2.3 Linux操作系统创建进程的方法

Linux内核实现了完善的进程机制，主要提供了fork、vfork 、clone三个进程创建方法，Linux源码中这三个调用的执行过程是执行fork()、vfork()、clone()时，通过一个系统调用表分别调用各自的内核函数sys\_fork()、sys\_vfork()、sys\_clone()，再在这三个函数中去调用do\_fork()函数进行具体的进程创建工作。

### 2.3.1 fork进程创建方法

fork函数无参数，当一个进程调用它并返回成功之后，程序就将变成两个进程，调用fork的进程为父进程，后来生成的为子进程。这两个进程将执行相同的程序文本，但却各自拥有不同的代码段、数据段以及堆栈拷贝。子进程的栈、数据以及栈段开始时是父进程内存相应各部分的完全拷贝，因此它们互不影响，例如子进程对父进程中的同名变量进行修改并不会影响其在父进程中的值。这样得到的子进程独立于父进程，具有良好的并发性。

fork函数接口头文件及函数原型如下所示：

#include<unistd.h>

pid\_t fork (void);

调用一次fork函数之后，返回两次值：在父进程中，fork()返回新创建子进程的进程ID；在子进程中，fork()返回0，利用其返回值的不同，可使父子进程执行不同的操作，fork函数的执行流程如图2.2所示。



图2.2 fork函数执行流程图

使用fork函数创建进程时存在以下两个问题：

进程创建成功后，父进程与子进程的执行顺序是无法确定的，在一些设计不好的程序中会导致资源竞争，从而出现不可预知的问题。可以在父进程中使用waitpid函数，等待其某个子进程终止后继续运行父进程来解决这个问题，图2.2 fork执行流程图中使用了该函数。

使用fork函数创建的子进程中对应的文件描述符与父进程均指向相同的打开的文件句柄，而且打开的文件句柄包含着当前文件的偏移量以及文件状态标志，所以在父子进程中处理文件时要考虑这种情况，以避免文件内容出现混乱或者别的问题。

### 2.3.2 vfork进程创建方法

vfork函数接口头文件与函数原型如下：

#include <sys/types.h>

#include <unistd.h>

pid\_t vfork(void);

vfork系统调用与fork基本是一样的，但用它创建的子进程并不完全复制父进程的虚拟地址空间。vfork创建的子进程与父进程共享地址空间，也就是说子进程完全运行在父进程的地址空间上，如果这时子进程修改了某个全局变量，这将影响到父进程。

由于vfork函数创建进程的过程消除了新的物理内存分配与虚拟地址空间复制的过程，因此其进程的创建更有效率。相对于vfork，使用fork时会导致的大量CPU占用，但是占用的时间是短暂的[24]。并且在现代的操作系统中，fork调用都使用了“写时复制”(Copy-on-write)技术，当父子进程中有更改相应段的行为发生时，再为子进程相应的段分配物理空间，减少不必要的开销，“写时复制”的实现如下：

1. 当一个进程创建时，它与父进程尽可能的共享同样的物理内存，内核仅仅复制进程的页表项并标明页的属性是“写时复制”；
2. 当进程去修改内存时，就会引发一个页异常，在异常的处理过程中，内核会分配新的物理页并复制要修改的内存，然后重新进行内存映射；
3. 当异常处理完毕返回后，进程修改的已经是新的物理内存，不会影响到原来与之共享内存的其他进程。

### 2.3.3 clone进程创建方法

fork与vfork进程创建方法是所有类UNIX系统都提供的传统的创建进程的系统调用，而clone是Linux系统独有的用于创建线程的系统调用方式，它可以用来创建进程或线程。从可移植的角度考虑，不鼓励直接使用 clone 系统调用。clone函数可以通过参数控制继承父进程的资源，因此创建方式比前面2种方法要复杂。同时该系统调用还能指定子进程的用户态堆栈指针。因此，这个系统调用函数既能创建用户进程，又能创建用户态线程[25]。

三种进程创建方法的比较如表2.3所示：

表2.3 三种进程创建方法对比

|  |  |  |  |
| --- | --- | --- | --- |
| 进程创建方法 | 可移植性 | 与父进程共享进程环境 | 选择继承资源 |
| fork | 良好 | 否 | 不支持 |
| vfork | 良好 | 是 | 不支持 |
| clone | 较差 | — | 支持 |

## 2.4 Eclipse平台及Java Swing用户界面开发技术

Eclipse是一种通用的工具平台，它支持开放式的可扩展的集成开发环境(IDE)。这个开放的开发环境，允许开发者可以高效地创建能够集成到Eclipse平台的工具[26]。Eclipse平台上的大部分功能的实现基于插件技术，如Java开发工具JDT(Java Development Tools)、插件开发环境PDE(Plug-in Development)等，功能如Workspace、Workbench等都是以插件的形式提供的。

Eclipse体系结构图如图2.3所示，它包括几个核心组件：workbench插件开发环境PDE、Java开发工具集JDT、Eclipse平台。PDE是提供专门开发插件的工具；JDT是提供了用于编辑、查看、编译、调试和运行Java代码的专门插件；Eclipse平台底层由标准Java2虚拟机支持，其核心是“Platform Runtime”微内核，基本作用是为工具提供方提供有利于无缝集成工具的使用机制和遵从的规则[27]。



图2.3 Eclipse平台体系结构图

通过用户图形界面GUI(Graphics User Interface)，用户和程序之间可以方便地进行交互[29]。Swing是一个用于开发Java图形界面应用程序的GUI(Graphics User Interface)组件，以抽象窗口工具包(AWT)为基础。Swing开发人员通过使用少量的代码，就可以利用Swing包中丰富、灵活的功能和模块化组件类来开发出令人满意的用户界面[30]。

抽象窗口工具集中有很多组件，Swing在其基础上对这些组件进行了进一步的修改，从而避免了早期使用AWT包开发时遇到的问题。Swing控件又被称为轻量级控件，这是由于Swing应用纯粹的Java代码对AWT的功能进行扩充，利用AWT中所提供的基本作图方法对树型控件进行模拟，并没有使用本地操作系统的内在方法来实现图形功能，因此方便移植到不同类型的平台上。

Swing包含250多个类、40多个组件，除了提供代替AWT的轻量组件外，还提供了大量有助于开发图形用户界面的附加组件[30]。使用Swing时需要注意以下两个问题：

避免在同一程序中混合使用Swing和AWT组件，减少异常发生的可能性。

Swing组件不是线程安全的[31]。如果我们试图通过多线程对用户界面中的各个Swing组件进行界面的刷新操作，我们的用户界面或JAVA控制台窗口将可能受到损坏[32]。需要自己设置同步机制，以避免破坏上述问题发生。

## 2.5小结

本章介绍了与本文研究内容相关的概念及技术。包括变异测试的基本思想、分析流程、方法优缺点的介绍，Linux下进程创建方法的对比以及Java Swing图形界面应用程序组件介绍。初步了解变异测试流程及相关术语以及fork函数的执行流程以及使用方法。

# 3基于并发机制的变异测试优化技术

本章节主要通过示例介绍基于并发机制的变异测试优化技术的基本原理、方法简述以及具体实现。

## 3.1基本思想

提出了一种基于并发机制的变异测试优化方法。变异测试产生的大量变异体使得变异体执行过程时间开销极大，该方法依据基本程序块以及软件并发机制，从变异体执行开销角度出发，探索在不影响变异测试故障检测能力的前提下提高变异测试效率。

与原始程序相比，变异测试产生的变异体在符合语法的前提下仅对待测程序作微小改动，因此简单程序经过变异算子演变后可产生大量变异体。一个程序块中通常包含一定数量的变异体，对于所属同一程序块的变异体执行相同的测试用例，在执行到变异代码之前，其程序的状态与原始程序完全一致（所走路径以及变量值）。

如图3.1所示是minmax程序的部分代码，所有变量初始化均为0。对于原始程序P，变异体M1在程序块i上有微小改动：变异体M1将“max = data[0];”变为“max = 8;”。



图3.1 minmax原始程序及变异体部分代码

输入测试用例{3,4,5,6,7}，在执行程序块i之前：

原始程序结果为：data={3,4,5,6,7}；min=0；max=0；size=0；i=0；t=5。

变异体M1结果为：data={3,4,5,6,7}；min=0；max=0；size=0；i=0；t=5。

执行程序块i后：

原始程序结果为：data={3,4,5,6,7}；min=3；max=3；size=5；i=1；t=5。

变异体M1结果为：data={3,4,5,6,7}；min=3；max=8；size=5；i=1；t=5。

由此可知，尽管变异体M1从程序入口处开始执行，在执行到变异代码所在程序块之前，其程序状态与原始程序相同。因此如果变异体可从变异代码所在程序块开始执行，即从程序入口到变异代码所在程序块之间的代码段仅执行一次，则可以节省部分执行时间。**基于并发机制的变异测试优化方法的基本思想就是考虑缩减从程序入口到变异代码所在程序块之间代码段的执行次数来提高变异测试效率。**利用该方法进行测试的过程中，变异代码所在程序块的前序程序块仅执行一遍，变异代码所在程序块及其后续程序块处理为并发分支，并发执行，以缩短变异体执行时间。

假定基于程序静态分析技术得出原始程序P分块结构为{B1,B2,…,Bi,…,Bn}，分析所有变异体集合在块上的分布情况，将变异体集合按照变异代码所属块进行分组。针对给定某个程序块Bi，与该程序块相关个变异体集合有K个{Mi,1,Mi,2,…,Mi,k}。图3.2从程序块的角度描绘了传统变异测试方法针对程序块Bi的变异体执行过程。



图3.2 传统变异测试方法针对程序块Bi的变异体执行过程

图3.3从程序块的角度描绘了基于并发机制的变异测试优化技术针对程序块Bi的变异体执行过程。通过对比可以更好的理解基于并发机制的变异测试方法原理，通过该技术，可以通过将程序入口到变异代码所在程序块之间代码段的执行次数缩减为1次，来减少变异测试执行过程中的时间开销。



图3.3 基于并发机制的变异测试优化方法针对程序块Bi的变异体执行过程

## 3.2方法简述

通过2.3节可知，多线程共享进程拥有的代码、数据、文件等资源，如果这时一个线程修改了某个全局变量，这将影响到其他线程，因此并不适用于本文所提及的变异测试优化方法。多进程状态下，子进程的栈、数据以及栈段开始时是父进程内存相应各部分的完全拷贝，因此它们互不影响，例如子进程对父进程中的同名变量进行修改并不会影响其在父进程中的值。这样得到的子进程独立于父进程，具有良好的并发性。因此多进程比多线程更适用于本文所提及的变异测试优化方法。

通过2.3 节linux进程创建方法对比可知，fork进程创建方法对于本文所提及的变异测试优化方法最优。**因此本文最终选定的优化方法为：基于Linux下的fork并发机制的变异测试优化方法。**

传统的变异测试方法采用所有的测试用例运行所有的变异体集合，通过比较变异体与原始程序的运行结果判定变异体是否被杀死，这无疑会耗费大量的计算资源。传统的变异测试流程如下图3.4所示，若存在测试用例t，在变异体m和原有程序p上的执行结果不一致，则称该变异体m相对于测试用例集T是可杀死变异体。



图3.4 传统变异测试流程

**基于并发机制的变异测试优化技术将变异体代码处理为并发分支，将从程序入口到变异代码所在程序块之间代码段的执行次数缩减为1次来提高变异测试效率。**图3.5详细描述了基于并发机制的变异测试优化技术流程：



图3.5 基于并发机制的变异测试优化技术流程

1. 假定基于程序静态分析技术得出原始程序P分块结构为{B1,B2,…,Bi,…,Bn}，分析所有变异体集合在块上的分布情况，将变异体集合按照变异代码所属块进行分组。
2. 针对给定某个程序块Bi，与该程序块相关的变异体集合有K个{Mi,1,Mi,2,…,Mi,k}。采用fork并发机制将原始程序P与{Mi,1,Mi,2,…,Mi,k}合成新的程序Pi’，将原始程序处理为父进程，发生变异的程序处理为一个并发分支。
3. 针对其他程序块分别执行步骤2，直到所有变异体均合并完毕。
4. 针对合成后的程序Pi’，对于给定的一组测试用例集合TS={T1,T2,…,Tj,Tl}，逐个执行测试用例集合中的每一个测试用例Tj，记录程序Pi’各个分支的执行结果，并与原始程序执行结果进行比较。如果结果不同，则对应变异体被检测出来，下一个测试用例无需再次执行该分支；否则，对应变异体存活，重发上述过程。直至所有与程序Pi’相关的变异体均被检测出来，或测试用例集合TS中的所有测试用例均被执行完毕，结束对于程序Pi’的测试。
5. 针对其他合并程序分别执行步骤4，直到所有合并程序均测试完毕，统计变异得分以及执行时间。

该技术需要解决的问题如下：

1. **如何创建多个子进程并控制进程执行顺序**

使用二元数组(fpid，count)控制进程的创建，其中fpid用来区分父、子进程；count用来控制创建进程的个数并标识子进程。

假设变异体个数为k+1个，在while循环中使用fpid = fork()创建子进程，控制循环条件为“fpid !=0 && count<k”，即进入该循环的条件为：该进程为父进程且count<k，使得父进程创建出k+1个子进程（count从0到k），并以count的数值标识该子进程为创建的第几个进程。

在while循环中加入waitpid(fpid, NULL, 0)函数控制进程的执行顺序，使得父进程等待进程组识别码与目前进程相同的子进程执行完毕后在执行。

其核心代码与执行流程如图3.6所示。

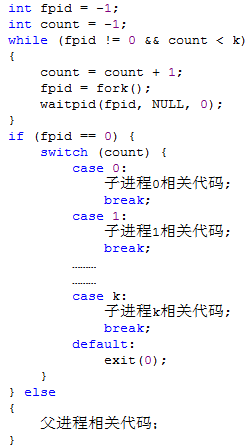


图3.6 多进程创建流程图

1. **如何将变异代码处理为并发分支**

实现了利用fork函数创建多个进程后，考虑如何将该框架与变异体相结合。fork函数创建子进程后，父子进程分别执行fpid>0与fpid=0条件下的代码，若无判断条件则父子进程均执行。例如如下代码，进程创建成功后父子进程将分别输出“a example of fork,fpid = 进程号”语句。

pid\_t fpid;

//若进程创建失败，则提示fork error

if ((fpid=fork()) == -1)

{

printf("fork error");

}

//进程创建成功后，有父子两个进程

//该处并未有父子进程判断条件，因此所有进程均执行该语句

printf("a example of fork,fpid = %d\n",getpid());

因此仅需将变异代码所在程序块写入多进程创建框架中的switch case多分支语句；将原始程序中相应程序块放入多进程创建框架中父进程相关代码处；在将该框架整体替换原始程序中相应程序块即可。示例程序原始代码及变异体m1、m2如表3.1所示，针对该程序的程序块处理后的程序代码如图3.7所示。

表3.1 示例程序及变异体代码

|  |  |  |
| --- | --- | --- |
| 原始程序 | 变异体m1 | 变异体m2 |
| void main() {  int a;  a=a+1;  printf("a=%d\n",a);  } | void main() {  int a;  a=a-1;  printf("a=%d\n",a);  } | void main() {  int a;  a=a+3;  printf("a=%d\n",a);  } |

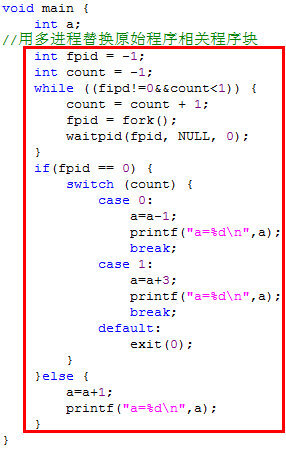
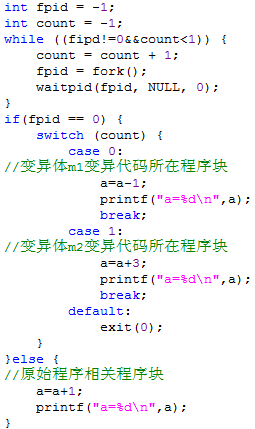


图3.7 多进程创建方法处理后的示例程序代码

1. **如何对应进程执行结果与变异体版本**

传统变异测试中变异体间的执行过程互不影响，因此可以通过输出重定向方法将变异体的执行结果保存到不同的文件中。在基于并发机制的变异测试优化方法中，变异体被合成一个多进程程序，变异代码所在程序块被处理为并发分支，若仅仅在测试执行过程中将执行结果重定向到文件，则所有变异体执行结果均保存到一个文件中，无法准确定位执行结果与变异体版本。因此考虑在变异体合成过程中，向并发分支代码中加入重定向输入输出流函数，使得每个变异体的执行结果均定向到不同文件中。重定向输入输出流函数如下所示：

freopen("mutantFileName","w", stdout)

该函数的函数声明如下所示：

FILE \*freopen(const char \* restrict filename, const char \* restrict mode, FILE \* restrict stream);

其中，参数filename表示重定向文件的名称，以变异体名称命名，用来区分不同变异体版本；参数mode代表文件访问权限的字符串，“w”表示“只写访问”；stream代表需要被重定向的文件流，“stdout”表示把标注输出重定向到一个文本文件中；定向成功，则返回该指向该输出流的文件指针，否则返回为NULL。

1. **如何避免重复执行已经检测出的变异体**

传统变异测试方法中，若执行某个测试用例后变异体与原始程序的输出结果不同，则该变异体被“杀死”，删除该变异体的可执行文件后，下个测试用例无需再次执行该变异体。

由于基于并发机制的变异测试方法将多个变异体合并为同一程序，测试用例执行合并后程序的可执行文件，无法达到单独删除某个变异体的可执行文件的目的。因此，基于并发机制的变异测试从变异体c语言文件考虑：若执行某个测试用例后变异体与原始程序的输出结果不同，则该变异体被“杀死”，删除该变异体的c语言文件。在合并程序中加入判断变异体.c文件是否存在的判定语句，如果存在则执行该分支，否则结束该分支所在进程，关键代码如下：

if (access("m1.c", F\_OK) == 0) {

变异体m1变异代码所在程序块语句

} else

exit(0);

基于并发机制的变异测试优化方法具有如下优点：

* **将多个变异体合成一个程序，减少的变异程序的编译次数。**
* **缩减了从程序入口到变异代码所在程序块之间代码段的执行次数。**
* **减少了无效测试用例的执行时间。**

示例程序以及进行程序合成后的支持并发机制的程序如图3.8所示。假设与该程序块相关的变异体共有m个，使用测试用例{2}去执行原始程序与变异体。可知，由于m=2，因此并不执行变异代码所在程序块，原始程序与变异体的输出结果并无不同，因此该测试用例为无效测试用例。

使用传统变异测试方法执行该测试用例需要执行m+1次（原始程序与m个变异体）。由于该测试用例并不执行case 0分支下的代码，因此在合并程序下使用基于并发的变异测试优化方法执行该测试用例时，并不执行进程创建函数以及各个版本的变异代码，极大程度上缩减了变异执行时间。

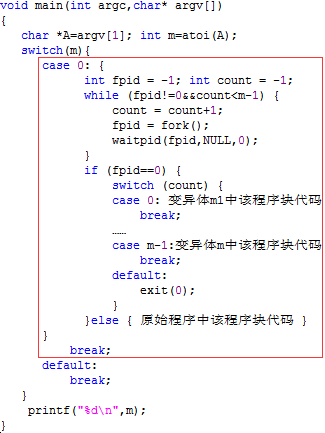
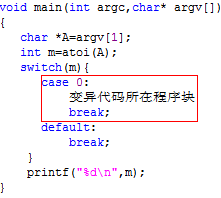


图3.8 无效测试用例示例代码

## 3.3具体实现

本节将主要介绍基于并发机制的变异测试优化技术的具体实现步骤和关键问题。图3.9具体描述了基于并发机制的变异测试优化技术的实现过程。



图3.9 基于并发机制的变异测试优化技术的实现过程

通过以下步骤能够实现基于并发机制的变异测试过程，具体步骤为：

1. 变异体分组：对比变异体与原始程序代码，定位变异代码位置，依据分组规则文件将变异代码行数对应到相关程序块，将变异代码行数所属同一程序块的变异体划分为一组。
2. 变异体合并：针对每个含有变异体的程序块，引入Fork机制将原始程序与该块相关的变异体集合合成一个程序，变异代码处理为并发分支。
3. 变异程序执行：执行合成后的程序，执行原始程序至发生故障前的所有程序块，并发执行该程序块相关的所有变异代码，依据原始程序与变异体的输出状态判定相应的变异体是否被杀死。

其中有三个关键问题：1) 如何定义分块规则；2) 如何将变异体集合与原始程序合并为支持并发机制的程序；3) 如何控制合并后程序的执行以及结果对比。

1. **分块规则定义：**

基于并发机制的变异测试方法首先需要分析相应程序块上的变异体集合分布情况，即按照变异体变异代码所在程序块对变异体集合进行分组，定义分块规则是该方法应用的关键问题之一。

使用“a~b”表示每个程序块的分块规则，其中a代表该程序块入口行数；b代表该程序块出口行数。分块规则文件包含待测程序的所有程序块的分块规则，分块规则所在行数代表该规则对应的程序块号。

1. **基于程序块的变异体合成技术：**

一个程序经过变异算子作用后生成大量变异体，一个程序包含多个程序块，每个程序块通常也会包含多个变异体，将大量变异体手动合成支持并发机制的程序是一件相当耗时的工作。为此，基于程序块的变异体自动合成技术是该方法应用的关键问题之一。该技术的主要步骤如下：

1. 分析分块规则文件，找到对应程序块的入口与出口语句行数，用二元数组(a,b)表示。关键代码如下：

**rule = ruleList.get(irowl-1).split("~");**

//取出程序块号irowl所对应的分块规则，将规则按“~”符号进行分割，并存储于字符串数组rule中。

**int a = Integer.parseInt(rule[0]);**//存储该程序块对应入口行数

**int b = Integer.parseInt(rule[1]);**// 存储该程序块对应出口行数

1. 针对某个程序块，基于该程序块的入口与出口语句行数，合并原始程序与相关变异体集合。

引入fork机制相关头文件，关键代码如下：

/\*添加fork函数头文件，使合成的程序支持fork机制\*/

**bw.write("#include <unistd.h>\n");**

**bw.write("#include <sys/types.h>\n");**

创建以程序块号为名称的C程序文件，将原始程序中除变异代码所在程序块的其他代码写入该文件中，引入多进程创建框架代替变异代码所在程序块，关键代码如下：

/\*创建以程序块号为名称的C程序文件\*/

**fw = new FileWriter("MyWorkSpace/" + programfileName + "/Block/" + blockDir[i] + "/" + blockDir[i] + ".c");**

bw = new BufferedWriter(fw);

/\*将原始程序代码保存至ArrayList<String>型的动态数组mutantsList\*/

**writeC("MyWorkSpace/Source/" + program);**

for(**int r=0;r<a-1;r++**)

{

/\*将变异代码所在程序块入口代码前的代码写入新程序\*/

**bw.write(mutantsList.get(r) + "\n");**

}

**/\*\***

**引入多进程创建框架，具体实现在本小节后续讲解**

**\*/**

for(**int r=b;r<mutantsList.size();r++**)

{

/\*将变异代码所在程序块出口代码后的代码写入新程序\*/

**bw.write(mutantsList.get(r) + "\n");**

}

bw.close();

引入多进程框架时，需将变异代码所在程序块写入多进程创建框架中的switch case多分支语句；将原始程序中相应程序块放入多进程创建框架中父进程相关代码处。

引入多进程编程关键代码：

/\*引入多进程创建框架\*/

bw.write("{\n");

bw.write("intfpid = -1;\n");

bw.write("int count = -1;\n");

// blockSub.length表示与该程序块相关的变异体集合个数

**bw.write("while (fpid != 0 && count<" + (blockSub.length-1) + ") {\n");**

bw.write("count = count + 1;\n");

bw.write("fpid = fork();\n");

bw.write("waitpid(fpid, NULL, 0);\n");

bw.write("}\n");

bw.write("if(fpid == 0) {\n");

bw.write("switch (count) {\n");

循环与该程序块相关的所有变异体，将变异代码所在程序块写入多进程创建框架中的switch case多分支语句：

for(int r=0;r<blockSub.length;r++){

bw.write("case "+ r +":\n");

/\*判断是否执行该并发分支，blockSub[r]表示变异体名称\*/

**bw.write("if ( access(\"" + blockDir[i] + "/" + blockSub[r] +"\",F\_OK) == 0 ) {\n");**

/\*将变异体输出重定向到以变异体名称blockSub[r]命名的文件中\*/

**bw.write("if(freopen(\"../Result/temp/"+blockSub[r]+".txt\",\"w\",stdout )==NULL)\n");**

bw.write("fprintf(stderr, \"e\\n\");\n");

bw.write("fflush(stdout);\n");

writeC("MyWorkSpace/"+programfileName+"/Mutation/"+blockDir[i]+"/"+blockSub[r]);

/\*将变异代码所在程序块写入合成程序，变异体处理为并发分支\*/

for(**int k=a-1;k<b;k++**)

{

/\*将变异代码所在程序块写入新程序\*/

**bw.write(mutantsList.get(k) + "\n");**

}

bw.write("}\n");

bw.write("else\n");

bw.write("exit(0);\n");

bw.write("break;\n");

}

将原始程序中对应变异代码所在程序块写入新程序，处理为父进程：

bw.write("default :\n");

bw.write("exit(0);\n");

bw.write("}\n");

bw.write("}\n");

bw.write("else {\n");

/\*将原始程序输出重定向到以Source命名的文件中\*/

**bw.write("if ( freopen(\"../Result/temp/Source.txt\",\"w\",stdout )==NULL)\n");**

bw.write("fprintf(stderr, \"e\\n\");\n");

/\*将原始程序代码保存至ArrayList<String>型的动态数组mutantsList\*/

**writeC("MyWorkSpace/Source/" + program);**

for(int r=a-1;r<b;r++)

{

/\*将原始程序中对应变异代码所在程序块写入新程序，处理为父进程\*/

**bw.write(mutantsList.get(r) + "\n");**

}

bw.write("}\n");

bw.write("}\n");

1. **基于程序块的变异体执行技术：**

变异体合并完成后，需要对合并后的程序进行编译、执行与测试结果评估。

编写shell脚本对合并后的程序进行编译、执行与测试结果评估过程及实现如下：

1. 获取合并程序名称，对其进行编译。

/\*去除路径，获取合并程序名称\*/

temp\_file\_b=` basename $file\_b `

/\*gcc 编译命令，生成可执行文件\*/

gcc -o ../Block/${temp\_file\_b}/${temp\_file\_b} ../Block/${temp\_file\_b}/${temp\_file\_b}.c

1. 从测试用例集中选取测试用例，判断与该合并后程序相关的变异体是否全部被检测出来。若没有全部被检测出来，则执行该合并程序；否则跳出，进行下一个合并程序的编译执行过程。

/\*统计存储与正在执行的合并程序相关变异体的文件夹内文件个数\*/

**count=$(ls ${temp\_file\_b} |wc -l)**

/\*若个数不为0，则说明还有变异体未被检测出来\*/

**if [ ${count} != 0 ]**

then

/\*执行测试用例temp\_file\_a \*/

while read line

do

../Block/${temp\_file\_b}/${temp\_file\_b} ${line}

done < ../Testcase/${temp\_file\_a}

else

/\*否则跳出循环，编译执行其他合并程序\*/

**break**

fi

1. 测试结果评估：

测试结果评估过程首先需要判断针对某个测试用例是否生产原始程序的输出文件，若为产生输出文件则无需进行后续测试结果评估过程。这是由于在变异体合成的过程中，将与变异代码相关的原始程序中的程序块处理为父进程并将其输出重定向为以Source命名的文件中。如果执行合成程序后并未出现该文件，则说明所使用的测试用例并未执行与变异代码有关的语句分支，因此变异体与原始程序的执行结果不会出现差异。

/\*判断测试用例是否执行与变异代码有关的语句分支\*/

**if [ -e "../Result/temp/Source.txt" ]**

then

strSource=""

while read line

do

strSource=${strSource}${line}

done < ../Result/temp/Source.txt

else

/\*不执行则无需后续文件内容对比，选择其他测试用例执行该合并程序\*/

**continue**

fi

若产生原始程序的输出文件，则需对比变异体与原始程序输出文件内容。若结果不同，则说明该变异体被检测出来，删除相应变异体c程序文件；否则不做更改。

/\*变异体与原始程序输出结果对比\*/

for file\_c in ${folder\_C}/\*;do

temp\_file\_c=` basename $file\_c `

temp\_file\_c=${temp\_file\_c%\.\*}

strMutation=""

while read line

do

strMutation=${strMutation}${line}

done < ${file\_c}

if [ "${strMutation}" != "${strSource}" ]

then

/\*结果不同则说明变异体被检测出来，删除该变异体c程序文件\*/

**rm -f ${temp\_file\_b}/${temp\_file\_c}**

fi

done

## 3.4小结

本章详细介绍了基于并发机制的变异测试优化技术基本思想、执行流程以及具体实现过程。并发机制的变异测试优化方法将所属同一程序块的变异体集合合并为一个程序，不仅减少了变异体编译次数，同时缩减了从程序入口到变异代码所在程序块之间代码段的执行次数。在合成后的程序上执行不覆盖变异代码所在程序块的测试用例时，不经过任何并发分支并且跳过结果对比流程（因为该用例并不执行错误所在语句，因此变异体与原始程序执行结果必然一致），相当于仅执行一次原始程序，极大程序的缩减了无效测试用例的执行时间。

# 4基于并发机制的变异测试优化支持工具的设计与实现

本章介绍基于并发机制的变异测试优化支持工具的需求与架构及实现基于并发机制的变异测试优化支持工具过程中的关键问题，并演示如何使用该工具进行变异测试。

## 4.1需求分析

基于并发机制的变异测试技术工具——CMC (Concurrent Mutation for C programs)，是一款面向C程序的测试工具，部署在Linux系统下。其系统用例图如图4.1所示。



图4.1 CMC系统用例图

该工具的主要目的是为了方便使用所提出的基于并发机制的变异测试优化技术。该工具应提供如下功能：

* 分析程序内部函数：对编译待测程序后生成的ELF文件的信息进行解析，生成符号表。
* 变异体分组：按照程序基本块对变异体集合进行分组。
* 合并变异体：对分组后的变异体集合进行代码合并。
* 执行测试过程：运行测试用例来验证该技术的可行性与工具的有效性。
* 输出测试报告：进行测试结果对比，测试结束后，输出测试结果报告。

下面将对图4.1进行各个用例的具体描述。

1. 选择待测程序：

测试者提供待测的C程序。进行测试前测试者需要导入待测试程序，或在主界面选择已存在的待测程序，工具根据该程序生成一系列目录及文档。选择待测程序包含解析ELF文件、生成测试目录两个用例。

* 解析ELF文件：工具根据用户选定的程序filename.c，使用GNU编译器命令：gcc –c filename.c，编译成二进制文件filename.o；对该文件进行符号列表分析，生成包含程序函数，全局变量的报告文件。
* 生成测试目录：工具根据用户选定的程序，生成后续操作需要的目录。生成目录功能及其结构分别如表4.1、图4.2所示。

表4.1 工具目录名称及其功能

|  |  |
| --- | --- |
| 目录名称 | 目录功能 |
| Source | 存储待测试程序 |
| Mutation | 存储变异体集合 |
| Rules | 存储分块规则文件 |
| Block | 存储分组后的变异体集合与合并后的程序 |
| Testcase | 存储测试用例 |
| Scripts | 存储编译、执行所需的脚本 |
| Result | 存储测试结果 |
| Log | 存储测试日志 |

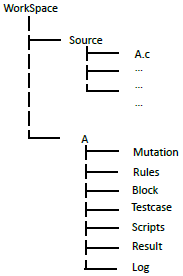


图4.2 工具目录结构

1. 对变异体进行分组：

选定待测程序后，测试者需要按照一定规则对选定待测程序的相应变异体集合进行分组。在进行分组的过程中，包括导入分块规则、导入变异体以及执行分组三个用例。

* 导入分块规则：用户需导入符合工具定义格式的程序分组规则文件。导入文件后可以在工具界面看到分组规则。
* 导入变异体集合：用户需导入与待测程序相关的变异体集合，导入后可在工具界面看到所有变异体名称。
* 执行分组：工具将自动对比变异体与待测程序变异行数，并对照分组规则将其分到相应组别。分组完成后，输出分组报告，包括共分为几组，每组包含哪些变异体，使得用户能够更加直观的了解分组情况。

1. 合成支持并发机制的变异程序：

分组完成后，工具将按程序块分组存放的变异体集合合并为支持fork并发机制的新程序，变异代码处理为并发分支，以组号命名新程序。合并完成后，输出合并报告，包括共合并为多少个程序，用户可查看每个程序的源代码。

1. 执行测试过程：

合并完成后，可调用shell脚本执行测试过程。在进行测试的过程中，包括导入测试用例集、执行测试脚本、测试结果评估、生成测试报告四个用例。

* 导入测试用例集：在执行测试脚本之前，用户需导入相关测试用例集。
* 执行测试脚本：导入测试用例集后，可执行测试过程。工具将调用shell脚本编译、执行合并后的变异程序。
* 测试结果评估：对比原始程序与变异体的输出状态判定相应的变异体是否被杀死。结果评估完成后，输出评估结果，包含变异体总个数、存活变异体个数、被检测出的变异体个数、变异得分、存活变异体名称以及被检测出的变异体名称。
* 生成测试报告：为了能够更好的记录每次测试后的结果，方便用户查看，测试执行完毕后，将会保存以“测试执行时的系统时间”命名的测试报告。报告包括：对于每个程序执行了哪些测试用例、测试结果保存名称、执行测试所花费的时间开销等信息。

1. 查看测试报告：

测试者可查看测试结果报告，获取更加全面的测试信息，包含对于每个程序执行了哪些测试用例、执行测试所花费的时间开销等信息。

## 4.2架构设计

针对需求分析，本节详细介绍基于并发机制的变异测试技术工具的工具架构、各组件的设计，工具的架构图如图4.3所示。



图4.3 CMC工具架构

图4.3描述了CMC工具的系统架构。使用灰色矩形框表示基本组件，对应待测程序选择器、变异体分组器、程序合并器以及执行与验证。每个组件由多个模块构成。本节对主要组件的功能及设计做简要概述。

1. 待测程序选择器

待测程序选择器，主要负责选定后续操作对应程序；生成相关测试目录；解析原始程序，给出程序函数及全局变量信息。

1. 变异体分组器

变异体分组器，主要负责解析分组规则文件，提取出代码行数与组别的转化规则；对比待测程序与变异体找出变异代码位置；依赖转化规则对变异体集合进行分组。

1. 程序合成器

程序合成器主要负责变异体与原始程序的合并。针对某个程序块将原始程序与该块相关的变异体集合合成一个程序，变异代码处理为并发分支。

1. 执行与验证

执行与验证，主要负责执行导入的测试用例，对比测试用例在原始程序与变异体执行后生成的结果；生成测试报告。

1. 查看报告

主要负责提供测试详细信息。

## 4.3工具的实现

按照工具功能要求，主要分为如下4个模块实现：

1. 待测程序选择模块

待测程序选择模块包含Main类，主要用于实现与选择待测程序相关的业务逻辑。主要方法如下：

* readSource()：在进入待测程序选择界面后调用该方法，用于读取工具MyWorkSpace/Source目录下存在的原始程序，显示在JTable控件上，便于后期提供给测试人员进行选择。
* readSub()：在点击“Import”菜单，选定需要导入的待测程序后调用该方法，用于将选定的待测程序导入到工具MyWorkSpace/Source目录下，使用tableModel.addRow()刷新JTable内容，便于后期提供给测试人员进行选择。
* creakMkdir(String destDirName)：在点击“select”按钮之后调用该方法，用于产生选定待测程序相关目录，为后期执行测试过程提供支持。

1. 变异体分组模块

变异体分组模块包含Divide类，主要用于实现与变异体分组相关的业务逻辑。主要方法如下：

* readRule(String path)：点击分组规则文件按钮后调用该方法，用来将测试人员选定的分组规则文件导入到Rules目录下，并在tabbedPane控件中显示文件内容。其中path为选定的文件目录。
* fileTraversal(String MutantsDir)：点击添加变异体集合按钮后调用该方法，用来遍历变异体集合，并将变异体名称显示在tabbedPane控件中。
* copyMutants(String fileName)：点击添加变异体集合按钮后调用该方法，将fileName命名的变异体导入到Mutation目录下，为后期分组、合并变异体做准备。
* boolean compareStr(String sourceStr, String mutationStr)：点击“Divide”按钮后调用该方法，对比变异体与原始程序内容，找到变异代码所在行数。
* int block(int row)：当compareStr函数返回的布尔值为真时调用该方法，读取分组规则文件，获取每个程序块的程序入口行数与出口行数，并判断第row行代码位于哪个程序块中，将其分组。

1. 变异体合并模块

变异体合并模块包含Commbine类、Shell类与Writeshell类。

Commbine类主要用于实现与变异体合并相关的业务逻辑，主要方法如下：

* combine()：点击“combine”按钮后调用该方法，对Mutation目录下文件夹进行遍历，获取分组个数；针对每个分组，将与该分组相关的变异体与原始程序合并成支持并发机制的新程序。
* Indent(String name)：对合并的程序进行格式化，调用Shell类中的myshell方法，执行C语言代码格式化命令，其中name代表需要格式化的文件名称。

Shell类主要实现与调用linux命令相关的业务逻辑，为其他类执行Linux命令提供方法。

Writeshell类主要实现与自动编写测试脚本相关的业务逻辑，为变异体执行提供脚本。

1. 变异体执行模块

变异体执行模块包含Run类与Shell类。

Run类主要用于实现与变异体执行相关的业务逻辑，主要方法如下：

* copyTestcase(String filename,int i)：点击添加测试用例“Add”按钮后调用该方法，将测试人员选择的测试用例filename按个数重命名，导入到Testcase目录下，为后期测试执行做准备。
* chmod()：为测试脚本设定运行权限。
* executeShell()：执行测试脚本，生成执行报告。
* totalMutants()：统计变异体集合总个数，显示在工具界面上。
* liveMutants()：统计存活变异体个数及名称、被检测出变异体个数及名称、变异得分，显示在工具界面上。

Shell类主要实现与调用linux命令相关的业务逻辑，为其他类执行Linux命令提供方法。

## 4.4工具演示

本节用nextdate程序作为示例，验证并演示CMC工具。

图4.4为待测程序选择界面。需要在界面提供的表单中选择待测程序，点击Select选择需要进行测试的程序，若列表中没有测试者需要的程序，则可点击菜单File导入。点击Next进入下一步。在Next动作执行时，工具会自动生成相应的测试目录，供后期测试使用。

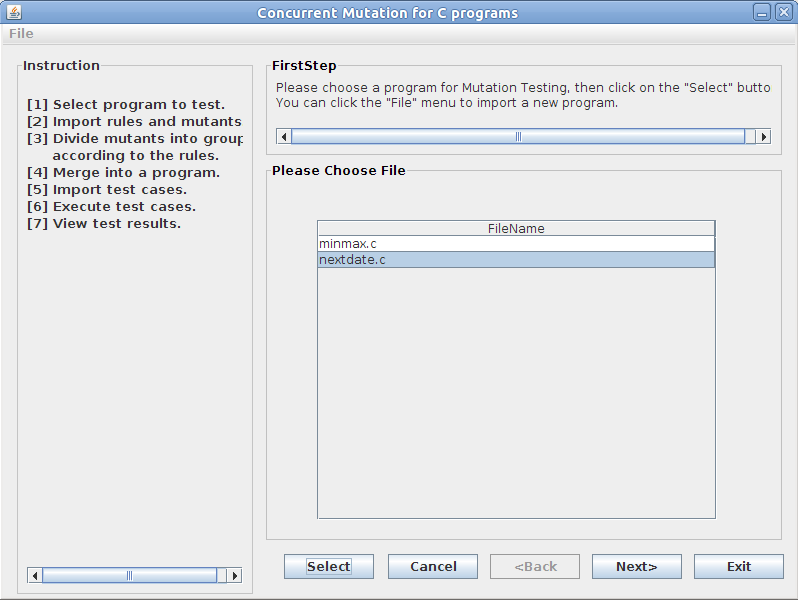


图4.4 待测程序选择界面

图4.5为变异体分组界面。针对变异体与原始程序代码位置的不同进行分组，点击Rules与Mutations下的Browse，浏览需要导入的规则文件与变异体集合并点击Add添加进测试目录。点击Divide对变异体集合进行分组，分组完成后，工具会反馈给测试人员具体的分组信息，包括变异体集合总个数，被分为几组，每组包含哪些变异体。

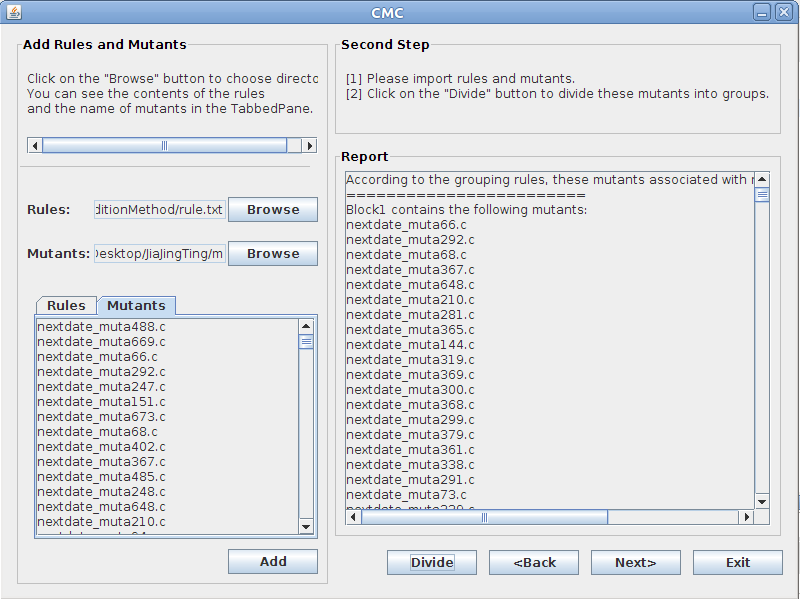


图4.5 变异体分组界面

图4.6展示了导入分组规则文件以及变异体集合后，工具反馈信息，包括分组规则内容以及所有变异体名称。

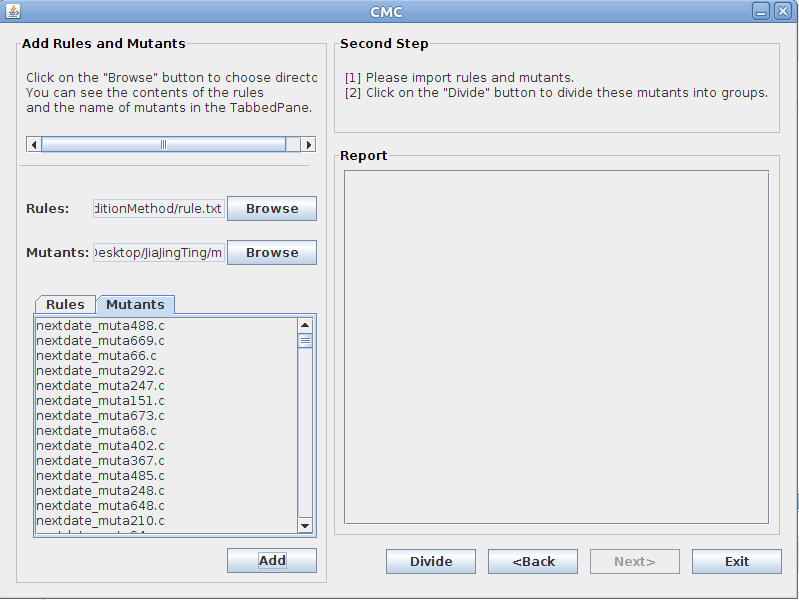
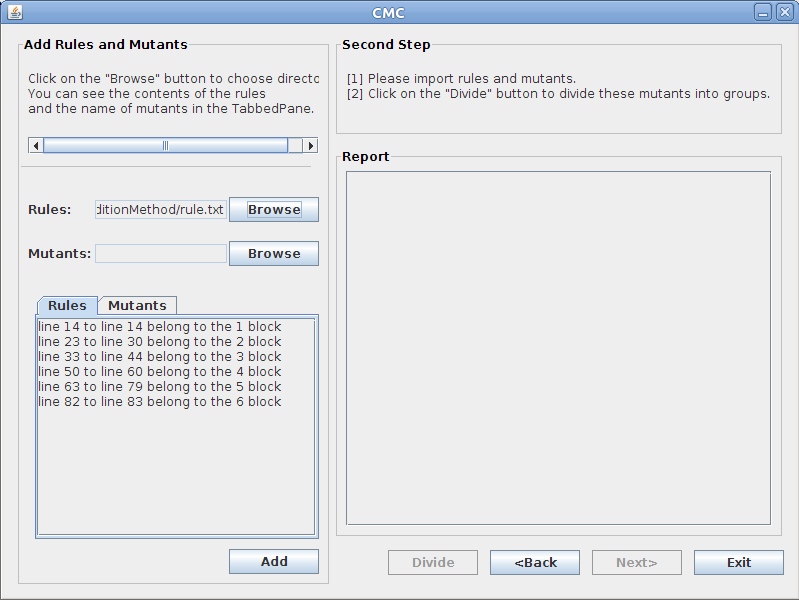


图4.6 导入分组规则文件以及变异体集合后的反馈信息

图4.7为变异体合并界面。工具对按程序块分组存放的变异体集合进行合并，合并为支持fork并发机制的新程序，变异代码处理为并发分支，以组号命名新程序。同时利用Indent代码格式化工具对合成的代码进行书写规范化。合并完成后，输出合并报告，包括共合并为多少个程序，测试人员可查看每个程序的源代码。

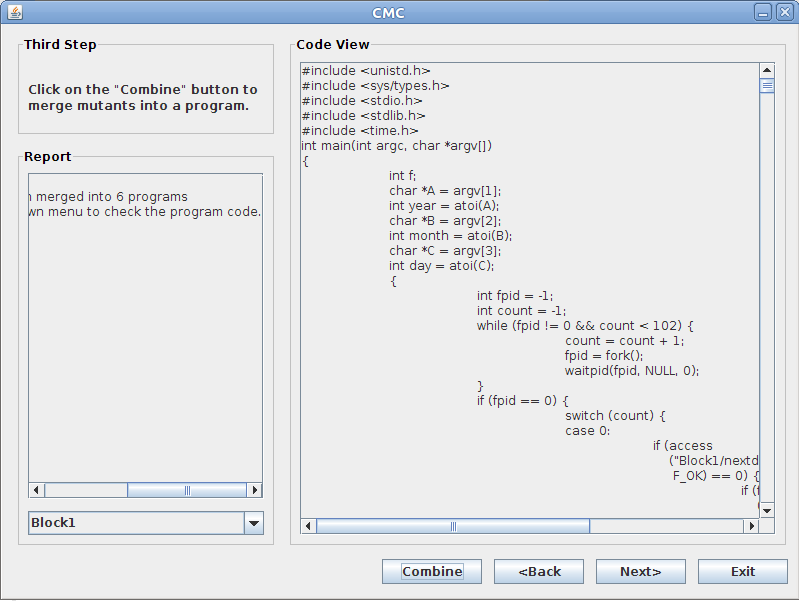


图4.7 变异体合并界面

图4.8为测试执行界面。点击Add导入测试用例集合后，点击Run后将执行相应测试脚本。测试执行完毕后，工具自动输出反馈信息，其中包括：变异体总个数、检测出的变异体个数及名称。存活的变异体个数及名称，以及变异得分。为了方便测试人员进行测试结果分析，工具会同时生成以以“测试执行时的系统时间”命名的详细测试报告，报告包括：对于每个程序执行了哪些测试用例、测试结果保存名称、执行测试所花费的时间开销等信息。测试报告内容及格式如图4.9所示。

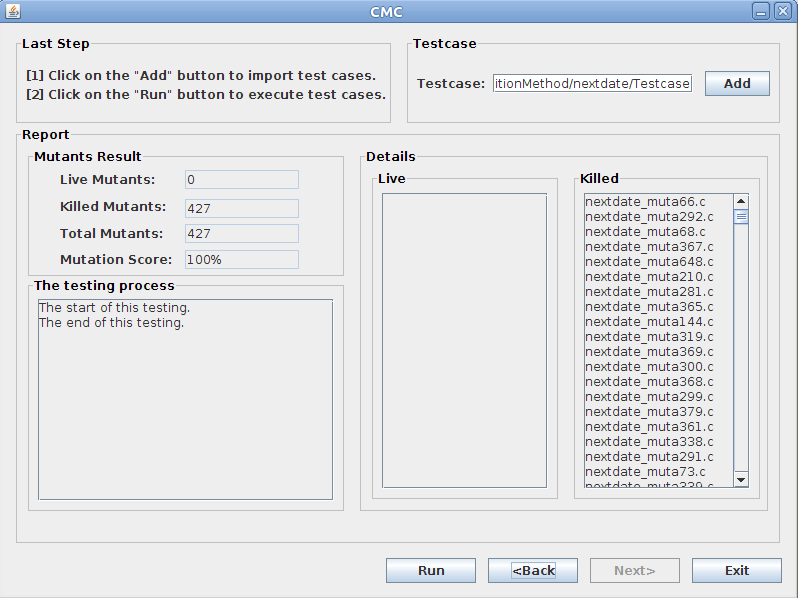


图4.8 测试执行界面

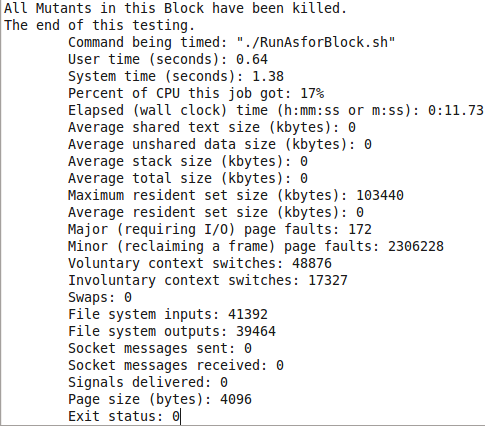
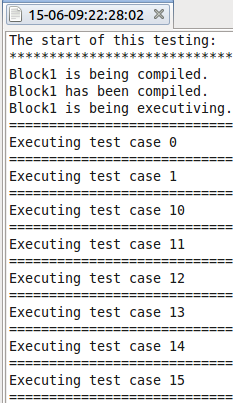


图4.9 测试报告内容及格式

## 4.5小结

本章讨论了基于并发机制的变异测试优化技术支持工具CMC的设计与实现，重点讨论了工具中待测程序选择模块、变异体分组模块、变异体合并模块、变异体执行模块的主要方法。最后以一组示例程序对演示了该工具的使用流程，展示了工具各个功能的具体使用方式。

CMC共涉及7个java类，总代码量2000多行。

# 5实验评估

本章选择两个C语言程序对基于并发机制的变异测试优化技术与支持工具进行实验评估。C语言程序源代码详见附录A、B。

## 5.1研究问题

本章的实验将围绕以下三个问题展开讨论。

1. 考察基于并发机制的变异测试优化技术与传统方法的测试执行时间开销。对于相同的变异体集合执行相同的测试用例集，记录两种方法的变异测试执行时间，为了防止偶然性的发生，进行5次实验取平均值。
2. 考察基于并发机制的变异测试优化技术的可行性与适用性。
3. 考察基于并发机制的变异测试优化技术对变异得分的影响。实验中将采用变异分析方法的度量指标变异得分(Mutation Score，MS)评估分析实验结果。

## 5.2度量指标

在本实验中使用**变异得分**与**变异测试执行时间**评估本文提出的基于并发机制的变异测试优化技术的有效性及工具的实用性。

变异得分定义了测试用例集检测出非等价变异体占变异体总数的百分比，若使用传统变异测试方法与使用基于并发机制的变异测试优化方法的变异得分相同，则说明优化方法并不影响变异得分。

变异测试执行时间包括：变异体执行、测试结果评估两个过程所花费的时间。对于相同的测试用例集与变异体集合，执行时间越少，说明该测试过程效率越高。

## 5.3实验对象

本节重点介绍实验对象的规格说明、输入输出描述、变异体、分组情况及测试用例规模描述等其他需要说明的信息。

### 5.3.1 minmax求最大值最小值程序

minmax最大值最小值程序针对输入的五元组整数进行大小对比，输出最小与最大整数。

1. 输入规格描述

输入用五元组(data[0], data[1], data[2], data[3], data[4])表示；代表需要进行大小对比的5个整数。

1. 输出规格描述

输出用二元组(min,max)表示；其中，min表示对比输入数组后找出的最小值；max表示对比输入数组后找出的最大值。

1. 变异体、分组情况及测试用例规模描述

针对用C语言变异测试工具Proteum生成的172个变异体进行处理，处理情况如表5.1所示。

表5.1 minmax程序变异体个数统计

|  |  |
| --- | --- |
| 变异体情况 | 变异体个数 |
| 变异体总数 | 172 |
| 多句变异个数 | 15 |
| 死循环 | 5 |
| 等价变异体 | 8 |
| 实验程序 | 144 |

使用处理后的144个变异体版本作为实验程序，测试该程序时所用测试用例规模为TC=40。分析相应程序块上的变异体集合，得到的分组情况如表5.2所示：

表5.2 minmax程序变异体分组情况统计

|  |  |
| --- | --- |
| 块号 | 变异体个数 |
| Block1 | 24 |
| Block2 | 29 |
| Block3 | 72 |
| Block4 | 19 |

### 5.3.2 nextdate求日期程序

nextdate求日期程序，针对输入的日期（包含年月日），按照闰年、平年每月天数规律求输入日期后一天的日期（包含年月日）。

1. 输入规格描述

输入用三元组(year, month, day)表示；其中，year代表输入年份；month代表输入月份；day代表输入天数。

1. 输出规格描述

输出用三元组(year, month, day)表示;其中，year代表输出年份；month代表输出月份；day代表输出天数。

1. 变异体、分组情况及测试用例规模描述

针对用C语言变异测试工具Proteum生成的487个变异体进行处理，处理情况如表5.3所示。

表5.3 nextdate程序变异体个数统计

|  |  |
| --- | --- |
| 变异体情况 | 个数 |
| 变异体总数 | 487 |
| 多句变异 | 24 |
| 死循环 | 0 |
| 等价变异体 | 36 |
| 实验程序 | 427 |

使用处理后的427个变异体版本作为实验程序，测试该程序时所用测试用例规模为TC=83。分析相应程序块上的变异体集合，得到的分组情况如表5.4所示：

表5.4 nextdate程序变异体分组情况统计

|  |  |
| --- | --- |
| 块号 | 变异体个数 |
| Block1 | 103 |
| Block2 | 55 |
| Block3 | 64 |
| Block4 | 60 |
| Block5 | 115 |
| Block6 | 30 |

## 5.4实验过程

为方便进行实验，采用shell脚本实现批量编译与执行变异体、测试结果验证、测试时间统计等功能。因此在实验前需设计支持实验流程的目录结构，使得编写的shell脚本能够正确运行。

传统变异测试方法实验目录结构如表5.5所示：

表5.5 传统变异测试方法实验目录

|  |  |
| --- | --- |
| 目录名称 | 目录功能 |
| Source | 存储待测试程序 |
| Mutation | 存储变异体集合 |
| Exe | 存储编译后的可执行文件 |
| Testcase | 存储测试用例 |
| Scripts | 存储编译、执行所需的脚本 |
| Result | 存储测试结果 |

基于并发机制的变异测试优化方法实验目录结构如表5.6所示：

表5.6 基于并发机制的变异测试优化方法实验目录

|  |  |
| --- | --- |
| 目录名称 | 目录功能 |
| Source | 存储待测试程序 |
| Mutation | 存储变异体集合 |
| Rules | 存储分块规则文件 |
| Block | 存储分组后的变异体集合与合并后的程序 |
| Testcase | 存储测试用例 |
| Scripts | 存储编译、执行所需的脚本 |
| Result | 存储测试结果 |

实例研究过程中，2个实验均是按照如下过程进行的：

1. 使用传统变异测试方法进行变异测试，统计变异得分，测试执行时间。具体步骤如下：

* 编译原始程序
* 编译变异体集合

利用shell脚本执行gcc编译命令，遍历所有变异体c程序文件，将其编译为可执行文件，编译脚本gcc.sh代码如下所示。使用命令time ./gcc.sh执行该脚本，记录变异体集合编译所花时间。

folder\_A="../Mutation" #设定变异体集合存储路径

for file\_a in ${folder\_A}/\*;do #遍历变异体集合

if [ ${file\_a##\*.} == "c" ]; then #针对c程序进行操作

temp\_file=` basename $file\_a `

file\_name=${temp\_file%\.\*} #获取变异体名称，不包括路径及拓展名

#编译变异体，在Exe目录下生成可执行文件

gcc -g -o ../Exe/${file\_name} ../Mutation/${temp\_file}

fi

done

* 执行测试用例并验证测试结果

导入测试用例到目录Testcase下，执行Scripts 目录下RunAsndiff.sh脚本，进行执行测试用例，验证测试结果。使用time ./ RunAsndiff.sh命令执行该脚本，记录执行测试用例与验证测试结果所花时间。

#传统变异测试方法下原始程序执行脚本

while read line

do

../../Source/nextdate ${line} > ../Result/${temp\_file\_a}/Source.txt

done < ../Testcase/${temp\_file\_a}

#传统变异测试方法下变异体执行脚本

if [ **${count} != 0** ] #如果仍有变异体存活，则继续执行测试用例

then

for file\_b in ${folder\_B}/\*;do #遍历存活变异体

temp\_file\_b=` basename $file\_b `

#执行存活变异体

while read line

do

../Exe/${temp\_file\_b} ${line} > ../Result/${temp\_file\_a}/${temp\_file\_b}.txt

done < ../Testcase/${temp\_file\_a}

done

else

**exit** #如果所有变异体均被检测出来，则结束此次测试

fi

#传统变异测试方法下测试结果检验脚本

for file\_c in ${testdir}/\*;do

temp\_file\_c=` basename $file\_c `

temp\_file\_c=${temp\_file\_c%\.\*}

string=""

while read line

do

string=${string}${line}

done < ${file\_c}

**if [ "${string}" != "${strSource}" ]** #对比输出结果

then

**rm -f ../Exe/${temp\_file\_c}** #结果不同删除该变异体可执行文件

fi

done

* 统计变异得分

1. 使用基于并发机制的变异测试优化方法进行变异测试，统计变异得分，测试执行时间。具体步骤如下：

* 变异体分组及合并

使用工具CMC进行变异集合的分组与基于程序块的变异体合并，生成支持并发机制的变异测试程序，并存储于Block目录下。

* 执行测试用例并验证测试结果

导入测试用例到目录Testcase下，执行Scripts 目录下RunAsforBlock.sh脚本，进行执行测试用例，验证测试结果。测试执行脚本RunAsforBlock.sh关键代码已在3.2小节一般过程中讲解（基于程序块的变异体执行技术），在此不做赘述。使用time ./ RunAsforBlock.sh命令执行该脚本，记录执行测试用例与验证测试结果所花时间。

* 统计变异得分

1. 为了防止偶然性的发生，重复5次步骤1、2，将执行时间取平均值。

## 5.5实验结果分析

统计并计算实验结果的变异得分及变异体执行时间，依据两个度量指标对基于并发机制的变异测试方法进行分析。分析将围绕5.1节的三个问题展开讨论。

### 5.5.1基于并发机制的变异测试优化技术与传统方法的测试执行时间对比

1. 实验对象：minmax比较大小程序

图5.1描述了传统变异测试方法与基于并发机制的变异测试优化方法对于minmax程序全部变异体5次的执行时间对比。从图中所示的结果显示，两种方法的执行时间均较为稳定；对于相同的变异体集合，基于并发机制的变异测试优化方法所花时间较少。

图5.1 minmax程序全部变异体执行时间对比

图5.2描述了传统变异测试方法与基于并发机制的变异测试优化方法对于minmax程序全部变异体平均执行时间对比，使用柱状图可以更加直观的看出两种方法执行时间的差异，实验结果表明对于minmax程序，基于并发机制的变异测试优化方法减少了50%的测试执行时间，优化方法能够有效减少测试所需时间。

图5.2 minmax程序全部变异体执行平均时间对比

图5.3至图5.6为对minmax程序4个程序块相应变异体的两种方法的执行时间对比，结果显示，基于并发机制的变异测试优化方法对于所属不同程序块的变异体集合在测试执行时间上均有不同程度的缩减。

图5.3 minmax程序Block1相关变异体执行时间对比

图5.4 minmax程序Block2相关变异体执行时间对比

图5.5 minmax程序Block3相关变异体执行时间对比

图5.6 minmax 程序Block4相关变异体执行时间对比

图5.7展示了传统变异测试方法与基于并发机制的变异测试方法下对minmax变异体分组执行的时间开销。从结果中可以看出，基于并发机制的变异测试优化技术能够有效降低块3的执行时间，显著消减相对于其他程序块执行时间开销较大的程序块的执行时间。

图5.7 minmax程序变异体分组执行平均时间对比

图5.8 比较了所属不同程序块的minmax变异体执行时间缩减比例，从实验结果可以看出，变异体程序所属块的执行顺序越靠后，变异体执行时间缩减比例越大。这是由于变异体程序所属块的执行顺序越靠后，父子进程分别执行代码越少，所需执行时间越少。

图5.8 所属不同程序块的minmax程序变异体执行时间缩减比例

1. 实验对象：nextdate求日期程序

图5.9描述了传统变异测试方法与基于并发机制的变异测试优化方法对于nextdate程序全部变异体5次的执行时间对比。从图中所示的结果显示，两种方法的执行时间均较为稳定；对于相同的变异体集合，基于并发机制的变异测试优方法所花时间较少。

图5.9 nextdate程序全部变异体执行时间对比

图5.10描述了传统变异测试方法与基于并发机制的变异测试优化方法对于nextdate程序全部变异体平均执行时间对比，使用柱状图可以更加直观的看出两种方法执行时间的差异，实验结果表明对于nextdate程序，基于并发机制的变异测试优化方法减少了65%的测试执行时间，优化方法能够有效减少测试执行时间。

图5.10 nextdate程序全部变异体执行平均时间对比

图5.11至图5.16为对nextdate程序6个程序块相应变异体的两种方法的执行时间对比，结果显示，基于并发机制的变异测试优化方法对于所属不同程序块的变异体集合在测试执行时间上均有不同程度的缩减。

图5.11 nextdate程序Block1相关变异体执行时间对比

图5.12 nextdate程序Block2相关变异体执行时间对比

图5.13 nextdate程序Block3相关变异体执行时间对比

图5.14 nextdate程序Block4相关变异体执行时间对比

图5.15 nextdate程序Block5相关变异体执行时间对比

图5.16 nextdate程序Block6相关变异体执行时间对比

图5.17展示了传统变异测试方法与基于并发机制的变异测试优化方法下对nextdate变异体分组执行的时间开销。从结果中可以看出，基于并发机制的变异测试优化方法能够有效降低块2和块5的执行时间，显著消减相对于其他程序块执行时间开销较大的程序块的执行时间。

图5.17 nextdate变异体分组执行平均时间对比

图5.18 比较了所属不同程序块的nextdate变异体执行时间缩减比例，从实验结果可以看出，变异体程序所属块的执行顺序与其执行时间缩减比例并未呈现出与minmax程序实验结果相同的规律。这是因为nextdate程序是以switch case多分支语句为主要程序结构的程序，并且程序本身执行时间较短。

如果选取的测试用例并不覆盖该变异代码所在程序块，则传统变异测试方法需在所属该程序块的所有变异体上均执行测试用例，并且进行结果对比。使用基于并发机制的变异测试优化方法时，由于该测试用例并未覆盖变异代码所在程序块，因此并未执行进程创建语句，相当于仅在原始程序上执行一次测试用例，并且无结果对比，这无疑会极大限度的减少测试时间。

因此对于分支结构较多的程序，其执行时间缩减比例与选取哪些测试用例以及测试用例的执行顺序有很大的关系。

图5.18 所属不同程序块的nextdate程序变异体执行时间缩减比例

### 5.5.2基于并发机制的变异测试优化技术的可行性与适用性

在本文基于并发机制的变异测试优化技术方法简述中详细描述了该方法的可行性，并在工具演示中进一步验证了该方法的可行性。下面分析下该方法对于不同程序结构的适用性。

按照结构化程序设计的观点，任何算法功能都可以通过由程序模块组成的三种基本程序结构的组合：顺序结构、选择结构和循环结构来实现。minmax程序主要以for/while循环结构为主，而nextdate主要以switch case多分支语句为主，顺序结构均出现在两个程序中。采用以上两个程序进行实验验证，证明了该方法对于c语言基本程序结构的适用性。

### 5.5.3基于并发机制的变异测试优化技术对变异得分的影响：

根据实验得到的两个例子的MS结果进行分析，考察上述方法所得到的故障检测能力。应用相同的测试用例集执行相同的变异体集合，实验结果显示，传统变异测试方法与基于并发机制的变异测试优化方法检测出minmax全部变异体使用了相同的9个测试用例；检测出nextdate全部变异体使用了相同的80个测试用例，两个程序测试的变异得分均为100%。结果证明使用基于并发机制的变异测试优化技术并不影响变异测试的故障检测能力。

## 5.6小结

通过两个实验研究对基于并发机制变异测试优化技术进行了验证与评估。从实验结果来看，基于并发机制变异测试优化技术适用于三种基本程序结构，在保障不影响变异得分的情况下，能够有效减少变异执行时间，提高变异测试效率。对于执行时间较长的程序块，优化效果更为明显。

# 结　　论

本文提出了一种基于并发机制的变异测试优化技术与支持工具。提出的技术与工具能够保证在不影响变异得分的情况下有效降低变异测试执行过程的时间开销。利用c程序评估所提方法的有效性以及工具的实用性。实验结果表明，基于并发机制的变异测试优化技术能够有效降低变异体执行过程中的时间开销。

本文的主要成果总结如下：

* 将变异测试与程序分析相结合，提出了基于并发机制的变异测试优化技术。
* 设计算法、编写脚本实现了基于程序块的变异体分组与变异体合并方法。
* 设计与实现基于并发机制的变异测试优化支持工具CMC，运用工具实现了变异体分组、变异体合并、执行测试用例、测试结果验证、生成测试报告等功能。
* 设计实验验证基于并发机制的变异测试优化方法的有效性及其支持工具的实用性。

进一步研究工作包括：

* 进一步优化支持工具，在实现基本功能的前提下，提高工具的操作友好性以及加入进度条、实时显示等交互功能。
* 进一步研究基于并发机制的变异测试优化技术的有效性，拓展实验程序使之对于多模块复杂程序进行验证。

# 参 考 文 献

1. Zhu H, Hall P A V, May J H R. Software Unit Test Coverage and Adequacy[J]. ACM Computing Surveys, 1997, 29(4): 366-427.
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. 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.
4. 姜凡, 郑人杰. 软件测试中的弱变异方法与关系测试数据[J]. 计算机学报, 1990, 13(8): 583-590.
5. 单锦辉, 高友峰等. 一种新的变异测试数据自动生成方法[J]. 计算机学报, 2008, 31(6): 1025-1034.
6. 陆毅明. 基于代数式规范的变异测试方法的设计与实现[J]. 计算机应用与软件, 2009, 26(2): 111-113.
7. Zhang Lu, Hou Shanshan, Hu Junjue, et al. Is Operator-Based Mutant Selection Superior to Random Mutant Selection?[C]. Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa, ACM Press, 2010: 435-444.
8. 徐拾义. 降低软件变异测试复杂性的新方法[J]. 上海大学学报:自然科学版, 2007, 13(5): 524-531.
9. 孙昌爱, 王冠. MujavaX：一个支持非均匀分布的变异生成系统[J]. 计算机研究与发展, 2014, 51(4): 874-881.
10. 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.
11. Mathur A P. Performance, Effectiveness, and Reliability Issues in Software Testing[C]. Proceedings of the 5th International Computer, Software and Applications Conference (COMPSAC 1991), Tokyo, Japan, 1991: 604-605.
12. Offutt A J, Lee A, Rothermel G, et al. An Experimental Determination of Sufficient Mutant Operators[J]. ACM Transactions on Software Engineering and Methodology, 1996, 5(2): 99-118.
13. Jia Y, Harman M. Constructing Subtle Faults Using Higher Order Mutation Testing[C]. Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation(SCAM 2008), Beijing, China, 2008: 249-258.
14. Polo M, et al. Decreasing the cost of mutation testing with second-order mutants[J]. Software Testing, Verification and Reliability, 2009, 19(2): 111-131.
15. Howden W E. Weak Mutation Testing and Completeness of Test Sets[J]. IEEE Transactions on Software Engineering, 1982, 8(4): 371-379.
16. Krauser E W, Mathur A P, Rego V J. High Performance Software Testing on SIMD Machines[J]. IEEE Transactions on Software Engineering, 1991, 17(5): 403-423.
17. Grun B J M, Schuler D, Zeller A. The Impact of Equivalent Mutants[C]. Proceedings of the 2009 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2009), Denver, Colorado, 2009: 192-199.
18. 茆亮亮. 变异测试技术应用研究[D]. 安徽: 中国科学技术大学, 2010.
19. 姚香娟. 基于路径比较的变异测试方法[J]. 电子学报, 2012, 40(1): 103-107.
20. 黄玉涵. 降低变异测试代价方法的研究[D]. 安徽: 中国科学技术大学, 2011.
21. 刘新忠, 徐高潮等. 一种基于约束的变异测试数据生成方法[J]. 计算机研究与发展, 2011, 48(4): 617- 626.
22. Gopinath R, Jensen C, Groce A, Code Coverage for Suite Evaluation by Developers[C]. Proceedings of 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, India, ACM Press, 2014: 72-82.
23. 严冰, 刘加海, 季江民等. Linux程序设计[M]. 杭州: 浙江大学出版社, 2012: 170-271.
24. 周丽焦, 程波, 兰巨龙. LINUX系统下多线程与多进程性能分析[J]. 微计算机信息, 2005, 21(9): 118-120.
25. 徐炜, Linux内核的进程创建和执行[J]. 电脑编程技巧与维护, 2006, 9:31-39.
26. 王津涛, 孟庆昌.Eclipse SWT/JFace开发实战精解[M].北京:人民邮电出版社,2007:1-10.
27. 魏楚元等. Eclipse:基于插件的下一代通用集成开发环境[J]. 计算机应用与软件, 2005, 22(6): 38-40.
28. 那静. Eclipse SWT/JFace 核心应用[M]. 北京: 清华大学出版社, 2007: 20-26.
29. 耿祥义. Java基础教程[M]. 北京: 清华大学出版社, 2009: 142.
30. 王鹏, 何昀峰. Java Swing图形界面开发与案例详解[M]. 北京: 清华大学出版社, 2008: 1-4.
31. 朱福喜. Java程序设计进阶教程[M]. 北京: 清华大学出版社, 2009: 1-6.
32. 曹大有, 王启东. JAVA 2 EventQueue类与JAVA 2 Swing组件的界面刷新[J]. 微计算机应用, 2006, 27(3): 376-378.

# 附 录

## 附录A 外文文献

Code Coverage for Suite Evaluation by Developers

Rahul Gopinath, Oregon State University, Corvallis, OR, USA, gopinath@eecs.orst.edu

Carlos Jensen, Oregon State University, Corvallis, OR, USA, cjensen@eecs.orst.edu

Alex Groce, Oregon State University, Corvallis, OR, USA, agroce@gmail.com

* ABSTRACT

One of the key challenges of developers testing code is determining a test suite’s quality – its ability to find faults. The most common approach is to use code coverage as a measure for test suite quality, and diminishing returns in coverage or high absolute coverage as a stopping rule. In testing research, suite quality is often evaluated by a suite’s ability to kill mutants (artificially seeded potential faults). Determining which criteria best predict mutation kills is critical to practical estimation of test suite quality. Previous work has only used small sets of programs, and usually compares multiple suites for a single program. Practitioners, however, seldom compare suites — they evaluate one suite. Using suites (both manual and automatically generated) from a large set of real-world open-source projects shows that evaluation results differ from those for suite-comparison: statement (not block, branch, or path) coverage predicts mutation kills best.

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

General Terms: Measurement, Verification

Keywords: Test frameworks, evaluation of coverage criteria, statistical analysis

* 1 INTRODUCTION

The purpose of software testing is to improve the quality of software, and the primary route to this goal is the detection of faults. Unfortunately, the problem of finding all faults in a program (or proving their absence), for any meaningful program, is essentially unsolvable. Testing is therefore always a trade-off between the cost of (further) testing and the potential cost of undiscovered faults in a program. In order to make intelligent decisions about testing, developers need ways to evaluate their testing efforts in terms of their ability to detect faults. The ability, given a test suite, to predict whether it is effective at finding faults is essential to rational testing efforts.

The ideal measure of fault detection is, naturally, fault detection. In retrospect, using the set of defects discovered during a software product’s lifetime, the quality of a test suite could be evaluated by measuring its ability to detect those faults (faults never revealed in use might reasonably have little impact on testing decisions). Of course, this is not a practical method for making decisions during development and testing. Software engineers therefore rely on methods that predict fault detection capability based only on the suite itself and the current version of the software under test (SUT). The most popular method is the use of code coverage criteria [1]. Code coverage describes structural aspects of the executions of an SUT performed by a test suite. For example, statement coverage indicates which statements in a program’s source code were executed, branch coverage indicates which branches were taken, and path coverage describes (typically in a slightly more complex way, to account for loops) the paths explored in a program’s control flow graph.

In software testing research, the gold standard for suite evaluation is generally considered to be actual faults detected, but this is, again, in practice difficult to apply even in a research setting [16]. The second most informative measure of suite quality is usually held to be mutation testing [7,2], which measures the ability of a test suite to detect small changes to the source code. Mutation testing subsumes many other code coverage criteria [29], and has been shown to predict actual fault detection better than other criteria in some settings [13, 2, 24], but never shown to be worse than traditional code coverage measures.

This paper examines the question of coverage criteria as suite quality predictors from the perspective of the nonresearcher audience, developers interested in suite evaluation (rather than comparison), lightweight, widely available tools, and well-known coverage criteria. Given the constraints under which real software projects operate; which widely available coverage criteria provide the best estimation of fault detection? This paper draws from the evaluation of hundreds of open source projects. While the results are based only on open source Java programs hosted on Github, and using the popular Maven build system, it is likely that our findings apply at minimum to many other Java projects, and may well apply to other languages as well. As a “silver standard” for evaluating suite quality, mutation testing is used, as identifying real faults of hundreds of Java projects was clearly infeasible.

Our findings based on real-world conditions show that, in contrast to the results of some studies conducted in a research context, statement coverage is generally the most effective predictor of suite quality. This is not an accident of the nature of manually-produced test suites; the same relationship also holds for test suites generated by the Randoop tool [31], which uses feedback-directed random testing to generate suites. While branch coverage or some variant of path coverage may be most useful for many research contexts, in the context of typical Java open source projects, at least, a focus on the simple and easily understood measure of statement coverage is probably most useful for predicting suite quality, even if developers are using an automated testing tool.

The primary contributions of this paper are twofold: First, the existence of popular open source repositories makes it possible to investigate the effectiveness of coverage criteria in a more unbiased, large-scale, and systematic way than previous studies. Such repositories provide a large body of very different SUTs, limiting selection effects. Using actual test suites from real projects also ensures that results are relevant to actual testing practices. The additional availability of automated testing tools mature enough to apply to this set of projects enables us to draw conclusions about both human-generated and automatically-generated suites, and to show that results do not depend on this property of test suites. This enables our second contribution: a practical proposal to developers wishing to evaluate test suites for open source Java projects. Finally, by showing that the best criteria for research purposes differ from those for practitioners, this paper shows that the preferences and abilities of software testing researchers may lead to less-than-optimal advice to developers whose focus is not on evaluating testing methods but on producing quality software.

* 2 RELATED WORK

A large body of work considers the relationship between coverage criteria and fault detection. The most closely related work to ours, which considers some of the same questions from a different perspective (that of researchers) is the recent work of Gligoric et al. [16]. Their work uses the same statistical approach as our paper, measuring both τβ and R2 to examine correlations to mutation kill for a set of criteria, and both studies consider realistically non-adequate suites. However, their work considers only a set of 15 Java programs and 11 C programs, selected not randomly but primarily from container classes used in previous studies and the classic Siemens/SIR subjects. Their larger projects (JodaTime, JFreeChart, SQLLite, YAFFS2) were chosen opportunistically. Our study is on a much larger scale in terms of subjects and uses a more principled selection process. Most importantly, however, we consider correlation of criteria across all SUTs, to answer the question “given a suite for an SUT, which criteria best predicts mutation kills for that SUT?” rather than to determine, within each SUT, which criteria best ranks various suites for that SUT.

Gligoric et al. report that branch coverage does the best job, overall, of predicting the best suite for a given SUT, but that acyclic intra-procedural path coverage is highly competitive and may better address the issue of ties, which is important in their research/comparison context. Inozemtseva et al. [21] investigates the relationship of various coverage measures and mutation score for different random subsets of test suites. They found that when the test suite size is controlled, only low to moderate correlation is present between coverage and effectiveness. This conclusion holds for all kinds of coverage measures used. The difference in subjects and focus yields substantially different results than ours, as we discuss below. Budd et al. [7] proposed mutation testing as a stronger criteria than other methods for evaluating test suites. Offut et al. showed that mutation coverage subsumes [29] many other criteria, including the basic six proposed by Myers [27]. Frankl et al. [13] compared the effectiveness of mutation testing with all-uses coverage, and found that at highest coverage levels, mutation testing was more effective. Andrews et al. compared [2] the fault detection ratio and the mutation kill ratio of a large number of test suites, finding that the ratios were very similar, and hence the faults induced by mutation representative of the real faults in programs. A follow up study [3] using a large number of test suites from a single program space.c found that the mutation detection ratio and the fault detection ratio are related linearly, with similar results for other coverage criteria (0.83 to 0.9). Linear regression on the mutation kill ratio and fault detection ratio showed a high correlation (0.9).

Li et al. [24] compared four different criteria (mutation, edge pair, all uses, and prime path), and showed mutationadequate testing was able to detect the most hand seeded faults (85%), while other criteria were similar to each other (in the range of 65% detection). Similarly, mutation coverage required the fewest test cases to satisfy the adequacy criteria, while prime path coverage required the most. Therefore, while there are no compellingly large-scale studies of many SUTs selected in a non-biased way to support the effectiveness of mutation testing, it is at least highly plausible as a better standard than other criteria.

Frankl and Weiss [12] performed a comparison of branch coverage and def-use coverage, showing that def-use is more effective than branch coverage for fault detection and there is stronger correlation to fault detection for def-use than branch coverage.

Gupta et al. [18] compared the effectiveness and efficiency of block coverage, branch coverage, and condition coverage, with mutation kill of adequate test suites as their evaluation metric. They found that branch coverage adequacy was more effective (killed more mutants) than block coverage in all cases, and condition coverage was better than branch coverage for methods having composite conditional statements. The reverse, however, was true when considering the efficiency (average number of test cases required to detect a fault) of suites.

Kakarla [23] and Inozemtseva [22] demonstrated a linear relationship between mutation detection ratio and coverage for individual programs. Inozemtseva’s study used machine learning techniques to come up with a regression relation, and found that effectiveness is dependent on the number of methods in a test suite, with a correlation coefficient in the range 0.81 ≤ r ≤ 0.93. The study also found a moderate-to-high correlation, with Kendall’s τ in the range 0.61 ≤ r ≤ 0.81 between effectiveness and block coverage when test suite size was ignored, which reduced when test suite size was accounted for. Kakarla found that statement coverage was correlated to mutation coverage in the range of 0.73 ≤ r ≤ 0.99 and 0.57 ≤ τ ≤ 0.94.

Wei et al. [35] examined branch coverage as a quality measure for suites for 14 Eiffel classes, showing that for randomly generated suites, branch coverage behavior was consistent across many runs, while fault detection varied widely. Early in random testing, where branch coverage rises rapidly, current branch coverage has high correlation to fault detection, but branch coverage eventually saturates while fault detection continues to increase; the correlation at this point became very weak.

Cai et al. [8] investigated correlations between coverage criteria under different testing profiles: whole test set, functional test, random test, normal test, and exceptional test. They investigated block coverage, decision coverage, C-use and P-use criteria. Curiously, they found that the relationship between block coverage and mutant kills was not always positive. They found that block coverage and mutant kills had a correlation of R2 = 0.781 when considering the whole test suite, but as low as 0.045 for normal testing and as high as 0.944 for exceptional testing. The correlation between decision coverage and mutation kills was higher than statement coverage, for the whole test suite (0.832), ranging from normal test (0.368) to exceptional test (0.952).

Namin and Andrews [28] also showed that fault detection ratio (non-linearly) correlated well with block coverage, decision coverage, and two different data-flow criteria. Their research suggested that test suite size was a significant factor in the model.

In general, none of this work considered a large, representative set of open source projects, and many studies considered the within-SUT suite comparison problem, not the problem of determining if a single suite provides effective testing for an SUT, as we do. The variety of reported rankings and correlations of criteria can be highly confusing to even a researcher wishing to compare suites, much less a typical (open source) developer seeking to decide if current testing for a project is effective for fault detection. Many studies do not even include all of branch, statement, and block coverage, the most readily available criteria. Our contribution over related work is a study that (1) uses a large set of open source projects, (2) uses both manually and automatically generated tests, (3) includes all the criteria of most interest to developers, and (4) focuses on the critical question of single-suite evaluation correlation.

* 3 METHODOLOGY

Our methodology was driven by two primary concerns: We wanted our results to be applicable to the largest set of real-world programs possible and based on a diverse set of actual test suites constructed by developers, not testing researchers. Our second concern was to strive for a statistically significant result, preferring to keep as many experimental variables constant as possible. One result of this constraint was to restrict the study to Java programs. Java is one of the most widely used programming languages [14, 34], and choosing a single language allows us to ensure a consistent definition for coverage criteria and avoid any difficulties due to variance in mutation operators. As a consequence, our results are only directly applicable to projects written in Java (a large portion of the code written today). However, the results are likely applicable to other programming languages with similar structures. Previous studies [16] do not show major differences between criteria effectiveness between Java and C programs, despite Java’s object-oriented nature, inclusion of exceptions, the different kinds of programs that tend to be written in C and Java, etc. Inferring criteria effectiveness for projects not based on C-like languages such as Java, C, C++, and C# — e.g. for functional languages—would be less justified.

Projects were taken from Github [15], one of the largest public repositories of Java projects. As a concession to ease of analysis, only projects using the popular Maven [4] build system were considered. Github provided an initial set of 1,733 projects meeting this criterion. While far from the entire set of Java projects hosted by Github, there is no reason it should be biased in terms of test suites. Note that it is also not the full set of Maven projects, since Github only returns 99 pages of search results. The process used by Github to select projects is not public, but we believe it is orthogonal to our concerns, and likely based on popularity and recency. After eliminating projects aggregating multiple projects (which are difficult to properly analyze), a set of 1,254 projects remained.

In order to ensure that our results remain free of systematic bias, we conducted our analysis in two phases. In the first phase, test suites present in the projects (mostly manually produced, though there may have been some automatically generated tests included) were used for coverage and mutation analysis. In the second phase, we generated test cases using Randoop [30], and performed the same statistical analysis using these suites. Finally, we compared the results of our first phase to that of the corresponding analysis in the second phase. Our aim in conducting this cross-validation was to ensure that our results would not be affected by possible bias in manually generated suites. Automatically generating suites allowed us to have a second, independent measure.

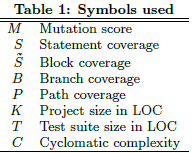
In both phases, we gathered coverage metrics from Emma [32] (statement coverage), Cobertura [11] (statement coverage, branch coverage), CodeCover [33] (statement coverage, branch coverage), JMockit [25] (statement coverage, path coverage), and PIT [10] (mutation kills). PIT is a tool aimed at developers rather than researchers, actively developed and supported, with some penetration in open source testing. While it is not explicitly mentioned in the JMockit project page, the path coverage provided by JMockit is similar to the Acyclic Intra Method Path (AIMP) coverage that performed well in the study by Gligoric et al. [16].

Out of our projects, only 729 projects had test suites. These were selected for the first phase of analysis. The analysis included running each of the coverage tools over the selected projects, with some effort expended on fixing trivial errors. In places where the compilation did not succeed after some effort, we discarded the project. Further, we specified a maximum timeout of 1 hour per project for a single tool. In the end, using original test cases we had 318 results from Cobertura, 286 results from Emma, 253 results from CodeCover, 361 from JMockit, and 259 from PIT.

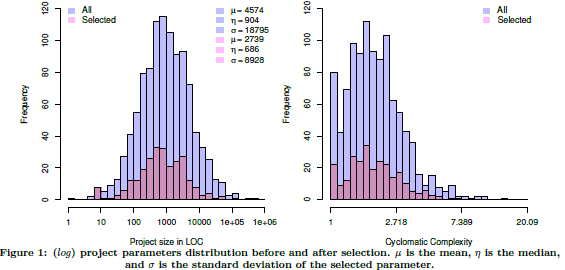
As an example of the selection process, consider mutation testing. Starting from 729 projects, 273 had compilation errors, dependency resolution problems, language version problems, or other fundamental issues preventing a build. Of the remaining projects, 37 timed out, requiring more than an hour for mutation testing with PIT. An additional 102 projects had test failures that prevented PIT from running, and PIT failed to produce any output for 39 projects, possibly due to no coverage or mutants produced, even though the build completed successfully.

For the second phase of analysis, we used Randoop, a feedback-directed random testing tool, on each of the projects, and discarded those where it failed to complete successfully or timed out. This produced test suites for 437 projects. From these suites, following the previous procedure, we obtained 314 results from Emma, 323 results from Cobertura, 287 results from CodeCover, 329 results from JMockit, and 243 results from PIT.

From the 437 projects for which we were able to generate Randoop suites, 66 again had compilation, etc. errors preventing analysis. Of the remaining, 4 projects timed out after over an hour with PIT, and 84 had test failures that prevented PIT from running. PIT failed to produce output for 51 of the Randoop suites. The symbols used to indicate various metrics collected are given in Table 1. The use of for block coverage is motivated by the observation that block coverage is a kind of weighted statement coverage; given source code and a CFG, block coverage can be computed given statement coverage details and vice-versa.



Since the coverage and mutation process resulted in a rather drastic reduction in sample space, we compared the distributions for code size and complexity before and after selection to verify that our procedure did not inordinately skew the sample space in at least these dimensions. The size distribution histograms for both before and after selection is provided in Figure 1.



Another important dimension in which a bias could appear is the complexity of programs; perhaps rejection is much more common with more complex or simpler programs, which could bias results, since coverage metrics are intimately tied to code complexity (for very simple programs, e.g., statement and path coverage are quite similar). We measured McCabe cyclomatic complexity [26] which provides a measurement of program complexity by counting the number of linearly independent execution paths through a program. The distributions before and after selections are given in Figure 1. These graphs suggest that selection did not unduly bias the sample in these two key dimensions.

To account for the effects of nondeterminism, we ran each coverage measurement 10 times, and computed the average. We also made use of multiple tools, as noted above, to verify that the coverage reported was accurate — e.g., Emma, Cobertura, CodeCover and JMockit all give statement coverage. Further, Cobertura and CodeCover provided branch coverage, and JMockit provided path coverage. Thus, we could compare most coverages provided by most tools and ensure they had high correlation to other tools. Further, we could ensure that the tools were processing all classes and test cases by examining statement coverage results. This was important because early on, we found that JMockit was not including classes that were not covered by any tests in its calculation of coverage1. Further, we have also removed a few observations (11 in original suites, 14 in Randoop suites) where the statement coverage reported by other tools was zero, and mutation or path coverage was non-zero, as these indicate some incorrect value from some tool. Our dataset, which includes metrics for all projects before elimination of outliers, is available for perusal in Dataverse [17].

* 4. ANALYSIS

The purpose of our analysis is to determine which coverage criteria that are likely to be used by real-world developers best predict mutation kill ratios. Our analysis also considers project and test suite size and cyclomatic complexity to determine if these factors affect the utility of coverage criteria.

The scatter-plots for mutation kills and statement coverage for both original test suites and Randoop-generated test suites are shown in Figure 2, with 232 pairs for original test suites, and 217 pairs for generated test suites. Similarly, the scatter-plots for mutation kills and branch coverage for both original test suites and generated suites is given in Figure 3, with 223 pairs for original test suites, and 191 pairs for generated test suites. The scatter-plots for path coverage in Figure 4 contain 214 pairs for original test suites, and 183 pairs for generated test suites. Finally, the scatter plots between block coverage and mutation kills are given in Figure 5. The diameter of the circles in all scatter plots correspond to the magnitudes of the project sizes (log(K)). The central result of these experiments is generally visible in these plots: statement coverage appears to give the best prediction of mutation kills of all criteria developers are likely to use, and this holds for both original and generated test suites.

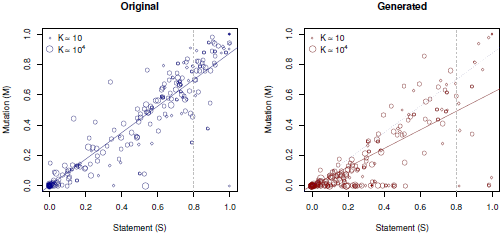


Figure 2: Relation between Statement Coverage and Mutation Kills. The circle represents the magnitude of project size.

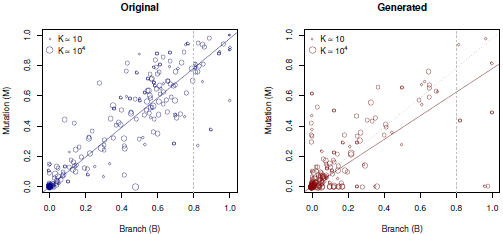


Figure 3: Relation between Branch Coverage and Mutation Kills. The circle represents the magnitude of project size.

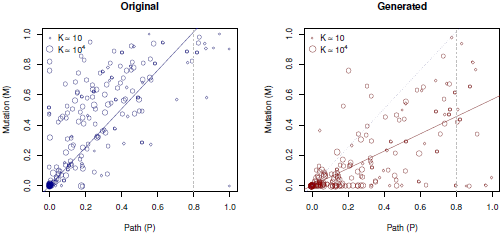


Figure 4: Relation between Path Coverage and Mutation Kills. The circle represents the magnitude of project size.

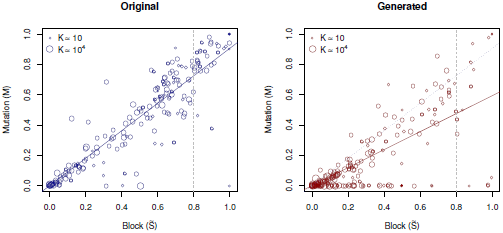


Figure 5: Relation between Block Coverage and Mutation Kills. The circle represents the magnitude of project size.

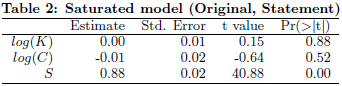
We use regression analysis and significance testing to ascertain the contribution of different factors to test suite effectiveness. The correlation coefficient R2 indicates the effectiveness of a model, i.e how much of the variation found in data is explainable by the parameters of the model. The factors that were found insignificant were eliminated to obtain reduced models.

**4.1 Statement Coverage and Mutation Kills**

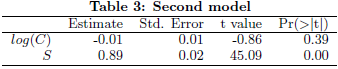
In this section, we try to find the significant factors that, in combination with statement coverage, predict mutation kills with a high degree of confidence.

We begin with the saturated model consisting of all variables. These are mutation score, project size in LOC, test suite size, cyclomatic complexity, and statement coverage. We removed the test suite size to avoid multicollinearity with project size after noticing that it correlated very strongly with the project size. Performing the same analysis with test suite size in place of project size gave the same results as below, except that the weak effect for branch coverage become a stronger effect (but for other criteria, suite size did not matter). This gives the regression relation:

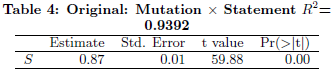
was set to zero since we had sufficient coverage data near the zero point, and zero statement coverage should indicate zero mutation coverage too. This is given in Table 2.



Further, we noticed that project size itself did not have a significant contribution to the response variable. Once we removed project size, our table was updated to Table 3.



Since cyclomatic complexity was also clearly not significant, removing it resulted in the equation and the result of this equation is in Table 4.



There is no significant effect of project size or program complexity on mutation coverage in a model based on statement coverage.

**4.2 Branch Coverage and Mutation Score**

In this analysis (Table 5), we follow the same path we took for statement coverage with mutation coverage. Project size had a very weak evidence of having an effect on mutation coverage at p = 0.0672 when compared to statement coverage (the effect for suite size here was stronger at 0.0015). was set to zero since we had sufficient coverage data near the zero point, and zero branch coverage, again, should indicate zero mutation kills.



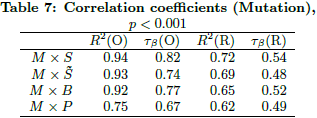
**4.3 Path Coverage and Mutation Score**

Following the same analysis steps (Table 6), we removed code size and cyclomatic complexity as they were not significant. is again set to zero for the same reasons.



**4.4 Comparing the Criteria**

After determining that project size, suite size, and cyclomatic complexity were essentially irrelevant for our purposes, we turned to comparing correlation statistics for all criteria for both original and Randoop-generated tests. In keeping with the most recent and extensive studies [16] we report both R2 and Kendall \_\_ correlations. R2 in our context is the most useful correlation measure, since ideally developers would like to predict the actual mutation killing effectiveness of a test suite. Kendall \_\_ is a rank-correlation statistic that is non-parametric, and therefore should be reliable even if underlying relationships are not linear — it aims to answer the question: given that the ranking between two coverage criteria for suites for projects is such that C(X) > C(Y), what is the chance that the ranking of mutation kills is in agreement with this ranking? The results from computing R2adj and τβ for each of the coverage metrics with mutation coverage are given in Table7. (O) indicates values for original test suites, and (R) indicates Randoop-generated suites.



The results are clear: across both original and generated suites, statement coverage has the best correlation for both R2 and τβ. For predicting mutation kills for test suites included with projects, branch, statement, and block coverage all provide a satisfactory method; predictions for Randoopgenerated suites are more difficult, but statement coverage still performs relatively well, with sufficient power to be useful in practice.

* CONCLUSION

Mutation testing is one of the best predictors of test suitequality, in terms of ability to detect actual faults. However, it is also computationally expensive to run, complex to apply, and is generally not used by real-world developers with any frequency. A long-term goal of the testing community has therefore been to develop alternative methods for predicting suite quality, for software development purposes and (perhaps primarily) for use in evaluating competing testing techniques. Unfortunately, the large body of previous studies on this topic have largely considered only a small set of programs, selected opportunistically, sometimes focused on coverage criteria used as rarely as mutation testing in real-world projects, and often been constructed around the question of predicting the best among multiple suites for a single SUT. In reality, software developers seldom have the luxury of applying esoteric coverage criteria or choosing between competing test suites. Rather, given an existing test suite, they want to estimate whether that suite is likely effective at detecting faults, or if more testing effort may be justified given the cost of faults.

This paper finds a correlation between lightweight, widely available coverage criteria (statement, block, branch, and path coverage) and mutation kills for hundreds of Java programs, for both the actual test suites included with those projects and suites generated by the Randoop testing tool. For both original and generated suites, statement coverage is the best predictor for mutation kills, and in fact does a relatively good (R2 = 0.94 for original tests and 0.72 for generated tests) job of predicting suite quality. SUT size, code complexity, and suite size do not turn out to be important. A simple model of mutation and mutation detection predicts the higher effectiveness of statement coverage, but does not explain why statement coverage even predicts path coverage better than branch coverage does, a highly counter-intuitive result.

The lesson for software developers is somewhat comforting: statement coverage is the most widely available and easily interpreted coverage criteria, and is also the best coverage criteria for predicting test suite quality in their context. The lesson for software testing researchers is that the question of how coverage correlates to suite effectiveness likely has no single correct answer, but must pay careful attention to the context of application, and the selection of a proper population of subjects and suites to examine.

* REFERENCES

1. P. Ammann and J. Offutt. Introduction to software testing. Cambridge University Press, 2008
2. J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In Proceedings of the 27th International Conference on Software Engineering, pages 402–411. IEEE, 2005.
3. 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):608–624, 2006.
4. Apache Software Foundation. Apache maven project. http://maven.apache.org.
5. T. Ball. A theory of predicate-complete test coverage and generation, 2004. Technical report, Microsoft Research Technical Report MSR-TR-2004-28.
6. T. Ball. A theory of predicate-complete test coverage and generation. In Formal Methods for Components and Objects, pages 1–22. Springer, 2005.
7. T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. Mutation analysis. Yale University, Department of Computer Science, 1979.
8. X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In ACM SIGSOFT Software Engineering Notes, volume 30, pages 1–7. ACM, 2005.
9. J. J. Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical report, DTIC Document, 2001.
10. H. Coles. Pit mutation testing. http://pittest.org/.
11. M. Doliner and Others. Cobertura - a code coverage utility for java. http://cobertura.github.io/cobertura.
12. P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transactions on Software Engineering, 19:774–787, 1993.
13. P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. Journal of Systems and Software, 38(3):235–253, 1997.
14. GitHub Inc. Github languages. http://www.github.com/languages.
15. GitHub Inc. Software repository. http://www.github.com.
16. M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In ACM International Symposium on Software Testing and Analysis. ACM, 2013.
17. R. Gopinath. Replication data for: Code coverage for suite evaluation by developers. In http: // dx. doi. org/ 10. 7910/ DVN/ 24574 . Harvard Dataverse Network V1, 2014-01.
18. A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. International Journal on Software Tools for Technology Transfer, 10(2):145–160, 2008.
19. M. M. Hassan and J. H. Andrews. Comparing multi-point stride coverage and dataflow coverage. In Proceedings of the 2013 International Conference on Software Engineering, pages 172–181. IEEE Press, 2013.
20. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In Proceedings of the 16th international conference on Software engineering, pages 191–200. IEEE Computer Society Press, 1994.
21. L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In Proceedings of the 2014 International Conference on Software Engineering, 2014.
22. L. M. M. Inozemtseva. Predicting test suite effectiveness for java programs. Master’s thesis, University of Waterloo, 2012.
23. S. Kakarla. An analysis of parameters influencing test suite effectiveness. Master’s thesis, Texas Tech University, 2010.
24. N. Li, U. Praphamontripong, and J. Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In International Conference on Software Testing, Verification and Validation Workshops, 2009. ICSTW’09., pages 220–229. IEEE, 2009.
25. R. Liesenfeld. JMockit - A developer testing toolkit for Java. http://code.google.com/p/jmockit/.
26. T. J. McCabe. A complexity measure. IEEE Transactions on Software Engineering, (4):308–320, 1976.
27. G. J. Myers. The art of software testing. A Willy-Interscience Pub, 1979.
28. A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In Proceedings of the eighteenth international symposium on Software testing and analysis, pages 57–68. ACM, 2009.
29. A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, 1996.
30. C. Pacheco and M. D. Ernst. Randoop random test generation. http://code.google.com/p/randoop.
31. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In Proceedings of the 29th International Conference on Software Engineering, pages 75–84. IEEE, 2007.
32. V. Roubtsov and Others. Emma - a free java code coverage tool. http://emma.sourceforge.net/.
33. R. Schmidberger and Others. Codecover – an open-source glass-box testing tool. http://codecover.org/.
34. TIOBE.Tiobeindex. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
35. Y. Wei, B. Meyer, and M. Oriol. Is branch coverage a good measure of testing effectiveness? In Empirical Software Engineering and Verification, pages 194–212. Springer, 2012.

## 附录B 外文翻译

从开发者角度考虑评估测试用例集的代码覆盖

**摘要**

开发人员测试代码的主要挑战之一是评估测试用例的质量——其发现故障的能力。最常见的方法是使用代码覆盖率来衡量测试用例的质量,并将一定覆盖率作为停止规则。在测试研究过程中,测试用例集的质量通常由杀死变异体(人工播种潜在故障后的程序)的能力来判断。确定哪些标准最好预测变异杀死是测试用例的质量实际估计的关键。以前的工作只使用了小程序集,并且针对单一程序通常使用较复杂的测试用例集。然而,从业者很少比较多个测试用例集——他们只评估一个测试用例集。使用从大量的真实的开源项目中手动和自动生成的测试用例集的结果表明：语句(不是块,分支,或路径)覆盖能最有效的预测变异体被杀死的概率。

分类和主题描述符：D.2.5(软件工程):测试和调试测试工具

一般条款：测量、验证

关键词：测试框架 覆盖标准评估 统计分析

* **1 简介**

软件测试的目的是提高软件的质量,这一目标的基本方法是故障检测。不幸的是,发现程序中的所有错误的问题(或证明他们的缺陷),对于任何有意义的程序,本质上是无法做到的。因此测试总是在测试所需成本和程序中未发现故障的潜在成本之间权衡。为了做出明智的测试决定,开发人员需要一些方法来依据故障检测能力评估他们的测试工作。给定一个测试用例集，预测其是否有效地找到故障的能力对合理的测试工作至关重要。

故障检测的最理想估量标准自然是故障检测本身。（使测量部件有测量自身能力的能力）回顾在一个在软件产品的生命周期中发现的一系列缺陷,一个测试用例集的质量可以通过测量其对于这些缺陷的检测能力进行评估 (从未发生过的缺陷几乎对测试决定没有影响)。当然, 在开发和测试期间这不是一个实际的决策方法。因此，软件工程师依赖仅基于测试用例集本身和被测试软件(SUT)的当前版本预测故障检测能力的方法。最受欢迎的方法是使用代码覆盖标准。代码覆盖率描述了测试用例执行时，被测试软件结构方面的表现。例如,语句覆盖率表明在程序中被执行的语句,分支覆盖显示哪些分支被执行,路径覆盖表明 (通常在一个稍微复杂的方式,考虑到循环)在程序控制流程图中的路径探索。

在软件测试的研究中, 评价用例的黄金标准通常被认为是实际故障检测率,但在实践中甚至在研究环境中很难应用。评价用例集质量的第二个最有效的方法通常是变异测试，通过检测出代码的微小变化能力衡量一个测试用例集。变异体测试包含了许多其他的代码覆盖率标准，并已被证明在一些情况下，比其他标准预测实际的故障检测效果更好,但从未显示出比传统的代码覆盖措施差。

不幸的是,变异测试很难被应用而且计算开销昂贵,这引发了寻找“更好”的预测测试用例集质量标准的研究。这一努力与真实软件开发人员高度相关，由于变异测试的复杂性、开销以及缺乏各种语言的支持工具，这些人几乎从未应用过变异测试。与研究人员不同的是，软件开发人员和测试工程师是从实际的角度开发软件的，然而，大多数有关评估测试用例集的研究不关注于他们的实际需要。

本文非研究员角度探讨的覆盖准则预测测试用例集的问题，开发者对使用轻量，广泛使用的工具，和常见覆盖准则评估测试用例集更有兴趣（而不是比较）。给定选择哪个真实项目的约束下，哪一个广泛使用的覆盖标准才是提供故障检测的最佳估计？本文利用数百的开源项目进行评价。虽然结果只是基于托管在Github上的开源Java程序，并采用了时下流行的Maven构建系统，很可能我们的研究结果最低适用于许多其他的Java项目，以及可能适用于其他语言。作为评估测试用例质量的“银标准”，应用变异测试识别数百个Java项目的实际故障检测率显然是不可行的。

基于现实世界的条件下，我们的研究结果表明，相比于一些基于研究背景下得出的结论，语句覆盖是关于软件测试用例集质量的测量中的最有效预测。这不是仅仅是因为手工生成测试用例集产生的偶然结果；同样这也适用于采用定向反馈随机测试生成测试用例的工具Randoop生成的测试用例集。虽然在典型的Java开源项目的背景下分支覆盖或路径覆盖的一些变种可能是最有用的，至少，即使开发者使用一个自动化测试工具，一个于简单又容易理解的方法——语句覆盖的是可能是预测测试用例集质量最有用的标准。

本文的主要贡献有两方面：首先，流行的开源库的存在使得有可能有更公正，大规模的并且比以前的研究系统化的方式研究覆盖准则效果。这样的存储库提供了非常多不同的被测试软件，限制了选择的影响。使用实际项目中的测试用例集也保证了结果符合实际测试实践。自动化测试工具的成熟，足以适用于这套项目，使我们得出针对手动生成的测试用例集与自动生成的测试用例集情况下的结果，结论表示结果不取决于测试用例集这个属性上。这是我们的第二个贡献，一个关于希望评估开源Java项目的测试用例集的实用建议。最后，通过展示用于研究目的的最佳标准与从那些从业者不同，本文表明，软件测试的研究人员喜好和能力可能会导致给予开发者（重点不在于评估测试方法而在于提高软件的质量的人）不理想的建议。

* **2 相关工作**

大量的工作与覆盖标准和故障检测率之间的关系有关。与我们最密切相关的，是近期的Gligoric 等工作，它参考了一些从不同的角度（即研究者）同样的问题。他们的工作与我们的论文使用了相同的统计方法，同时测量τβ和R2评估变异体检测率与一系列标准的相关性，这两项研究都考虑了现实不充分的用例。然而，他们的工作只考虑了一组15 Java程序和11个C程序，他们的挑选不是随随便便而是主要来源于以前研究的容器类和经典的西门子/ SIR项目。 他们研究的项目中其较大的项目（JodaTime，JFreeChart的，SQLLite，YAFFS2）是随机选出的。我们的研究面向一个规模更大的项目，使用了更原则性的选择过程。然而，最重要的是，我们考虑所有被测试软件的标准，以期望回答“针对被测试软件给出一个测试用例集，哪个标准预测变异体是否被杀死的效果最佳？”的问题，而不是决定，针对每一个待测软件，找到最合适该软件的测试用例集。

Gligoric等人指出对于给定的被测软件的分支覆盖的效果最好，但无循环的内部程序​​路径覆盖是高度覆盖的，可能会更好地解决关系问题，这是在他们的研究/比较重要的方面。Inozemtseva等人研究了各种覆盖方法与变异得分在测试用例集不同的随机子集中的关系。他们发现，当测试用例集的规模被控制，覆盖率和效力之间只呈现低到中度相关性。这一结论适用于所有类型的覆盖措施。

Budd等人提出变异测试作为一种比其他方法来评估测试用例集质量更强的标准。 Offut等人表明，变异体的覆盖涵括许多其他标准，包括Myers建议的基本的6个标准。

Frankl等人比较了变异体测试以及其他覆盖方法的有效性，并发现，在最高的覆盖水平，变异测试是更有效的。 Andrews等人将故障检测率和大型测试用例集的变异体检测率比较，发现该比率非常相似，得出变异体中包含的故障可代表程序的实际故障。随访研究使用了从单个C语言程序space中生成的大量测试用例，发现变异体检测率和故障检测率是线性相关的，与其它覆盖准则具有相似的结果（0.83到0.9）。变异体检测率和故障检测的线性回归分析表现出两者具有较高的相关性（0.9）。

Li等人比较了四种不同的标准（变异体，边对，所有用途，主要路径），并显示出变异体足够的测试能够检测最多人工故障（85％），而其他条件均彼此类似（在65％的检测范围）。同样，变异体覆盖所需的最少测试用例来满足充分性准则，而主要路径覆盖所需的最。因此，虽然没有大量研究来支持变异体测试的有效性，它至少是比其他标准更好的高度可信的标准。

Frankl和Weiss进行的分支覆盖率和定义—使用覆盖率的比较，显示出定义—使用覆盖标准比分支覆盖故障检测更为有效，并同故障检测率有较强的相关性。

Gupta比较了块覆盖、分支覆盖和条件覆盖的有效性和效率，有足够的测试套件作为其评价指标的变异体杀灭。他们发现，分支覆盖比在所有情况下的块覆盖更为有效（杀死的变异体数量更多），条件覆盖比分支覆盖具有复合条件语句的方法更好。

Kakarla和Inozemtseva证实了独立程序下变异体检测率和覆盖之间的线性关系。 Inozemtseva的研究使用机器学习技术提出了一个回归关系，结果发现在相关系数范围0.81 ≤r≤ 0.93，其效果取决于在一个测试用例集的方法数。研究还发现一个中度至高度的相关性，当测试用例集规模被忽略时，有效性和块覆盖的Kendall的τ的范围在0.61 ≤τ≤ 0.81。 Kakarla发现在0.73 ≤r≤ 0.99和0.57 ≤τ≤ 0.94的范围内语句覆盖率与变异体覆盖呈正相关。

Wei等人研究分支覆盖作为14个Eiffel类的测试用例的度量方法，显示出为随机生成的用例集，分支覆盖的行为在其运行上一致，而故障检测差别很大。早期随机测试，其中分支覆盖迅速上升，目前的分支覆盖与故障检测具有很高的相关性，而随着故障检测的持续提高，分支覆盖最终达到饱和，在这一点上的相关性变得非常弱。

Cai等人研究在测试的不同方面覆盖标准之间的相关性：整个测试集，功能测试、随机测试，正常测试和异常测试。他们调查了块覆盖，判定覆盖，计算使用准则和判定使用准则。奇怪的是,他们发现块覆盖率和变异体检测之间的关系并不总是呈正比的。在考虑整个测试测试用例集时，他们发现块覆盖率和变异体检测的相关性R2=0.781，但是正常测试情况下低至0.045，异常测试情况下高达0.944。、判定覆盖和变异体检测的相关性高于语句覆盖，对整个测试用例集情况下(0.832)，正常测试情况(0.368)到异常测试情况(0.952)。

Namin和Andrews表示，故障检测率（非线性）和块覆盖，判定覆盖，和两个不同的数据流标准有很好的相关性。他们的研究表明，测试用例集规模的大小在模型中是重要的因素。

在一般情况下，这些工作没有一个是大的，有代表性一套开源项目，许多研究认为被测试软件内部测试用例集合比较问题，而不是正如我们做的问题：确定一个单一的测试用例集合为被测试软件提供有效的测试。许多研究甚至不包括最容易获得的分支、语句和块覆盖标准。我们对相关工作的研究是（1）采用较大的开源项目集合，（2）采用手动和自动生成测试，（3）包括开发人员最感兴趣的的所有标准，（4）重点针对单个测试用例集评估相关的关键问题。

* **3 方法**

我们的方法基于两个主要的问题：我们希望我们的结果将最大限度的适用于由程序员编写而非测试研究的并基于大量多样化真实测试用例集的实际测试用例。我们的第二个关心的问题是争取一个统计学显着的结果，宁愿保持尽可能多的实验变量不变的可能。这种约束的一个结果是限制了对Java程序研究。 Java是最广泛使用的程序设计语言之一 ，选择一种语言使我们能够确保覆盖准则一致的定义，并避免由于变异算子的多样性而造成的任何困难。因此，我们的结果是仅直接适用于用Java编写的项目（在今天广泛使用的代码之一）。然而，结果很可能适用于具有类似的结构的其他编程语言。尽管Java的面向对象的性质本质上属于一种异类，以前的研究并没有显示出C和Java编程的主要差别，不同种类的程序，往往用C和Java等语言编写对项目的推断标准效益如果不是基于C模式语言如Java，C，C++和C＃。例如，功能性语言——将是不太合理的。

项目取自Github上，Java项目的最大的公共库之一。退一步以缓和分析，只考虑用时下流行的专业构建系统的项目。Github上提供的一组1,733项初始的项目符合这一准则。虽然远离整套通过Github上托管的Java项目，也没有任何理由应该在测试用例集方面失之偏颇。请注意，这也不是全套Maven项目，因为Github上只返回99页的搜索结果。使用Github上选择项目的过程是不公开的，但我们相信这是正是我们所关注的，并有选择可能受程序的欢迎程度和新旧程度影响。消除项目，整合多个项目（很难正确分析）后，一共还有1254项目仍然存在。

为了确保我们的结果仍没有系统偏差，我们分两个阶段进行我们的分析。在第一阶段，测试用例集目前在项目（主要是人工编制的，虽然有可能包括一些自动生成的测试用例）用于覆盖和变异分析。在第二阶段中，我们使用Randoop生成测试用例，并使用这些集合进行同样的统计分析。最后，我们比较了第一阶段与第二阶段的对应分析结果。我们在此进行交叉验证的目的是为了确保我们的结果不会在手动生成测试用例集带来的偏差受其影响。自动生成用例集允许我们有不一样、独立的方法。

在这两个阶段，我们从Emma（语句覆盖）的Cobertura (语句覆盖，分支覆盖）， CodeCover（语句覆盖，分支覆盖）, JMockit (语句覆盖，路径覆盖)以及 PIT（变异体检测）收集覆盖率指标。PIT是面向开发人员，而不是研究人员的一个工具。它被积极开发和支持，并在开放源代码的测试中有些渗透。虽然在JMockit项目页面未被明确地提及，由JMockit提供的路径覆盖方法类似于AIMP覆盖，这种覆盖在Gligoric等人在研究中效果良好。

我们的项目中，只有729个项目拥有测试用例集。这些被选作第一个阶段的分析。分析包括在每一种覆盖工具中运行选定的项目，修复某些错误。在花费很多努力后仍然编译不成功的工程，我们便放弃。此外，对于每个工具，我们指明每个程序最多运行时间为一个小时。最后，运行原装的测试用例情况下，我们有318个源于Cobertura 、286个源于Emma、253个源于CodeCover 、361个源于 JMockit 和259个源于PIT的结果。

作为选择过程的一个例子，考虑变异测试。从729个项目开始，有273个有编译错误，解决依赖问题，语言版本的问题，或防止构建等基本问题。其余的项目中，有37个超时，都需要用PIT做一个多小时的变异测试。一个额外的102个项目有测试失败，阻止PIT运行，并且PIT未能产生任何输出项目有39个，可能是由于尽管整体编译成功完成，但没有覆盖或变异体产生。

对于第二阶的段分析，我们在各项目中使用Randoop，一种反馈定向随机测试的工具，并弃去那些它未能成功编译或超时的项目。这437个项目生产了测试用例集。从这些集合中，按照前面的步骤，我们从Emma得到314个结果、 从Cobertura得到323个结果、从CodeCover得到287个结果、从JMockit得到329个结果以及从PIT得到243个结果。

从437个我们能够产生Randoop用例集的项目中，66又有了编译，等等，防止错误的分析。剩下的，4个项目在PIT上运行超过一个小时，84个已经测试失败，阻止PIT运行。PIT未能产生输出为Randoop用例集的项目51个。

用于指示收集各种度量的码元列表如表1所示。对于块覆盖使用表示，说明块覆盖是一种加权的语句覆盖。给定的源代码和控制流图（Control flow graph），块覆盖可以计算出语句覆盖信息，反之亦然。

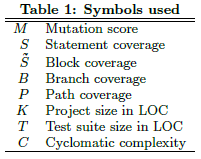


表1 各种标准的码元列表

由于覆盖和变异过程导致一个样本空间大幅减少，我们比较了代码大小和复杂性的分布之前和选择后，以验证我们的程序没有大幅度的改变样品空间中。图1展示了选择之前与之后的该粒度分布直方图。

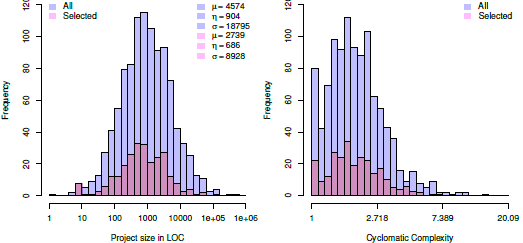


图1 项目参数分布前和选择后（μ平均值，η中位数，σ所选择的参数的标准偏差）

另一个重要方面，其中可能出现的偏置是方案复杂设置；因为覆盖度量与代码复杂度是紧密联系在一起的（非常简单的项目，例如，语句覆盖和路径覆盖是非常相似），这可能产生偏差。我们测量McCabe圈度复杂：一种通过程序计数的线性独立的执行路径的数量测量程序复杂的方法。在图1中给出后这些图表表明，选择没有过分偏向于这两个主要方面的样本。

考虑到不确定性的影响，我们对于每个覆盖测量10次，计算平均值。我们还使用了多种工具，如上面提到的，验证覆盖率报告是准确的。例如：Emma，Cobertura，Codecover和JMockit提供语句覆盖，Cobertura和Codecover提供分支覆盖，JMockit提供了路径覆盖。因此，我们可以比较大多数的工具提供的覆盖准则并且确保他们跟其他工具有很高的相关性。此外，我们能确保工具通过检查语句覆盖处理所有的类和测试用例的结果。此外，我们还删除了一些意见——在一些工具中得出语句覆盖率为零，但变异或路径覆盖是非零的，因为这些表明一些工具的一些不正确的值。我们的数据集，其中包括前消除所有项目指标异常，可在Dataverse中细读。

* **4 分析**

我们的分析的目的是确定哪些由开发人员使用的覆盖标准能够最好的预测变异体检测率。我们的分析还考虑项目、测试用例集的大小和圈复杂度，以确定这些因素是否会影响覆盖准则的效用。

变异检测和语句覆盖率在原始测试用例集和Randoop生成的测试用例集情况下的关系散点图如图2所示，包含232个原始测试用例，217个生成的测试用例。同样，变异检测和分支覆盖在原始测试用例集和Randoop生成的测试用例集情况下的关系散点图如图3所示，包含223对原始测试用例，和191对生成的测试用例。路径覆盖散点图如图4所示，包含214对原始测试用例和183对生成的测试用例。最后，图5给出了块覆盖和变异检测之间的关系。在所有的散点图中，圆的直径对应的项目规模的大小（log（k））。从这些散点图中可以显而易见的得出结论：无论对于原始测试用例集合还是自动生成的测试用例集，在所有开发者可能会用到的覆盖准则中，语句覆盖似乎给出了变异检测的最佳预测。

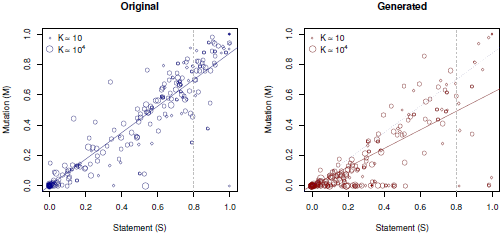


图2 ：语句覆盖和变异检测的关系（圆圈代表项目规模的大小）

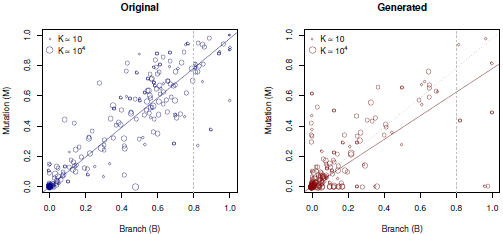


图3 ：分支覆盖和变异检测的关系（圆圈代表项目规模的大小）

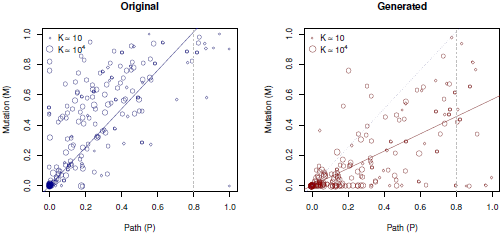


图4：路径覆盖和变异检测的关系（圆圈代表项目规模的大小）

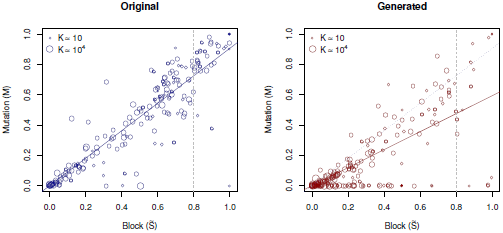


图5：块覆盖和变异检测的关系（圆圈代表项目规模的大小）

我们用回归分析和显著性检验，以确定不同的因素对测试用例集效力的贡献。相关系数R2表示模型的有效性。该因素淘汰了微不足道的因素，得到了简化模型。

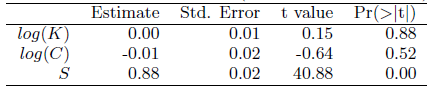
**4.1语句覆盖和变异检测**

在本节中，我们试图找到显著因素，与语句覆盖相结合，以高度的信心预测变异检测率。

我们从包括所有变量的饱和模型开始。它们是变异得分、项目软件规模代码行、测试用例集规模、圈复杂度和语句覆盖。在注意到它与项目规模有非常强烈的相关之后，我们移除了测试用例集规模这一变量，以避免项目规模的多重共线性。用测试用例集规模代替项目规模进行分析后，得出结果如下所示，除了对微弱的分支覆盖效果变得更强（但是对于其他的标准，用例集大小没有关系）。下面是回归关系：

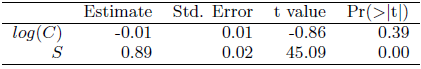
β0设置为0，因为我们有足够的覆盖数据接近零点，零语句覆盖率也应对应零变异覆盖。如表2所示。

表2：饱和模型（原始，语句）



此外，我们注意到，项目的规模本身并不对响应变量有显著贡献。一旦我们除去项目规模，我们的表就更新成了如表3所示。

表3：第二种模式



因为圈复杂度也显然不是特别显著，删除它导致的方程：

表达式的结果如表4所示。

表4：原文：变异体×语句R2=0.9392



对项目规模或者以语句覆盖率为基础的变异覆盖的模型程序复杂性没有显著影响。

**4.2分支覆盖和变异分数**

在这种分析中（表5），我们采取了跟我们研究语句覆盖率与变异覆盖时相同的方法。项目规模关于对变异覆盖的影响有一个非常微弱的迹象，相比于语句覆盖准则（这里的效果提高0.0015）P=0.0672。β0被设定为零，因为我们有接近零点的足够的覆盖数据，以及零分支覆盖，再次，应说明零变异检测。

表5：原始：变异X分支R2 = 0.9231



**4.3路径覆盖和变异体得分**

按照相同的分析步骤（表6），我们删除了代码大小和圈复杂度，因为它们不显著。β0又因为同样的原因而设置为零。

表6：原始：变异X路径R2=0.7496

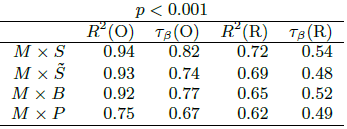


**4.4比较覆盖标准**

对我们而言，在确定项目规模，用例集大小和圈复杂度基本上是无关紧要的，我们转向比较在原始测试用例与Randoop生成的测试用例的情况下所有相关统计。在最新和最广泛的研究中，我们报告了两个R2和Kendall τβ相关性。 在我们的文章中R2是最有用的相关性度量，因为理想的开发者想预测在一个测试用例集中实际检测变异体的有效性。Kendall τβ是一种秩相关统计信息，非参数，而且因此应该是可靠的，即使底层的关系不是线性的——它是用来回答：鉴于两者覆盖标准的强度，对于项目的用例集是C(X)>C(Y)，变异检测的强度和该强度保持一致的机会是什么？

每种覆盖准则与变异覆盖计算出的的R 2和τβ的结果由表7给出。（O）表示原始测试用例，（R）表明Randoop生成测试用例。

表7：相关系数（变异），P <0.001



结果很明显：在原始和生成的用例集中，语句覆盖R2和τβ的结果最佳。分支覆盖标准、语句覆盖标准以及块覆盖标准均提供了较好的预测项目原始测试用例集检测变异体的能力的方法；预测Randoop 工具生成的测试用例模块检测变异体的能力比较困难，但语句覆盖仍然有较好的表现，在实践中有足够的应用。

* **结论**

就实际故障检测能力来看，变异测试是评估测试用例集质量的最佳指标之一。然而，它的运行成本很昂贵、计算开销大、应用复杂并且通常不被现实世界的开发人员使用哪怕一次。为了软件开发的目的，（也许为主）用于竞争性的测试评估技术，测试界的长期目标也因此变成开发替代预测测试用例集合质量的方法。不幸的是，前人就这个话题的研究已经很大程度上被认为只是针对一个小型程序集，有时关注在实际项目测试中很少用到的覆盖标准，并且往往是围绕针对单一的被测程序，哪个测试用例集效果最优的问题研究。事实上，软件开发人员很少运用深奥的覆盖标准或在测试用例集之间进行选择。而是，给定的现有的测试用例集合，他们要估计该集合是否可能有效地检测故障，或者是否进行更多的测试工作。

基于数百个Java程序实验，本文发现轻量的级，广泛使用的覆盖准则（语句、块、分支、路径覆盖）和变异检测之间的相关性，其中所用测试用例集包括那些项目原始测试用例集和由Randoop测试工具生成的测试用例集。对于这两种原始和工具产生的测试用例集，语句覆盖标准是用于预测变异检测的最佳标准，以及实际上做了相对较好（R2=0.94原始测试用例集下的实验结果和0.72工具生成的测试用例集下的结果）的预测测试用例集的质量的工作。被测试软件大小、代码的复杂性和测试用例集的规模不会很重要。变异和变异检测的一个简单模型预测了语句覆盖率更有效，但并不能解释为什么语句覆盖率甚至比分支覆盖率预测路径覆盖率更好。

这个课题对于软件开发者来说是有点安慰：语句覆盖是最广泛使用的，容易理解的覆盖标准，也是在其背景下预测测试用例集的质量最佳的覆盖准则。这个课题对于软件测试的研究人员的贡献是：关于覆盖准则与测试用例质量的关系可能没有唯一正确的答案，但一定要注意应用背景，以及选择合适的项目规模及测试用例集规模进行考察。

## 附录C minmax程序源代码

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main(int argc,char\* argv[])

{

int data[5];

int min;

int max;

int i,j,t,size;

for(t=0;t<5;t++) {

char \*A=argv[t+1];

data[t]=atoi(A);

}

min = data[0];

max = data[0];

size = 5;

i= 1;

while (i < size) {

if (data[i] < min) {

min = data[i];

}

if (data[i] > max) {

max = data[i];

}

i++;

}

printf( "%d %d\n",min,max);

return 0;

}

## 附录D nextdate源代码

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main(int argc,char\* argv[]) {

int f;

char \*A=argv[1];

int year=atoi(A);

char \*B=argv[2];

int month=atoi(B);

char \*C=argv[3];

int day=atoi(C);

f=((year%4==0)&&(year%100!=0))||(year%400==0);

switch(month)

{

case 1:

case 3:

case 5:

case 7:

case 8:

case 10:

if(day>=1&&day<31)

day=day+1;

else if(day==31) {

day=1;

month=month+1;

}

else

printf("ERROR!\n");

break;

case 12:

if(day>=1&&day<31)

day=day+1;

else if(day==31) {

day=1;

month=1;

year=year+1;

}

else {

printf("ERROR!\n");

exit(0);

}

break;

case 4:

case 6:

case 9:

case 11:

if(day>=1&&day<30)

day=day+1;

else if(day==30){

day=1;

month=month+1;

}

else

{

printf("ERROR!\n");

exit(0);

}

break;

case 2:

if(day>=1&&day<28)

day=day+1;

else if((!f)&&day==28){

day=1;

month=month+1;

}

else if(f&&day==28)

day=day+1;

else if(f&&day==29){

day=1;

month=month+1;

}

else

{

printf("ERROR!\n");

exit(0);

}

break;

}

printf("%d %d %d\n",year,month,day);

return 0;

}

## 附录E 实验原始数据

minmax程序全部变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 25m56.070s | 12m29.417s |
| 2 | 25m55.170s | 12m28.169s |
| 3 | 25m54.570s | 12m28.375s |
| 4 | 25m53.000s | 12m28.750s |
| 5 | 25m51.862s | 12m32.821s |

minmax程序Block1相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 2m3.458s | 1m20.618s |
| 2 | 2m5.052s | 1m21.421s |
| 3 | 2m3.177s | 1m20.094s |
| 4 | 2m3.000s | 1m20.056s |
| 5 | 2m3.038s | 1m20.142s |

minmax程序Block2相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 4m57.420s | 3m5.494s |
| 2 | 4m57.199s | 3m5.515s |
| 3 | 4m57.177s | 3m5.404s |
| 4 | 4m57.440s | 3m5.220s |
| 5 | 4m57.330s | 3m5.374s |

minmax程序Block3相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 16m47.617s | 5m52.502s |
| 2 | 16m46.166s | 5m54.178s |
| 3 | 16m51.789s | 5m51.884s |
| 4 | 16m49.826s | 5m53.680s |
| 5 | 16m42.722s | 5m51.525s |

minmax程序Block4相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 3m3.354s | 0m24.018s |
| 2 | 3m3.540s | 0m24.080s |
| 3 | 3m3.379s | 0m24.030s |
| 4 | 3m3.422s | 0m23.976s |
| 5 | 3m3.422s | 0m24.041s |

nextdate程序全部变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 0m29.614s | 0m10.242s |
| 2 | 0m27.655s | 0m9.511s |
| 3 | 0m28.671s | 0m10.801s |
| 4 | 0m28.792s | 0m9.333s |
| 5 | 0m27.850s | 0m9.289s |

nextdate程序Block1相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 0m9.444s | 0m5.627s |
| 2 | 0m9.922s | 0m5.097s |
| 3 | 0m9.920s | 0m5.479s |
| 4 | 0m10.037s | 0m5.362s |
| 5 | 0m9.469s | 0m5.073s |

nextdate程序Block2相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 0m6.103s | 0m0.804s |
| 2 | 0m5.733s | 0m0.781s |
| 3 | 0m5.762s | 0m0.853s |
| 4 | 0m5.544s | 0m0.793s |
| 5 | 0m5.381s | 0m0.706s |

nextdate程序Block3相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 0m2.369s | 0m0.763s |
| 2 | 0m2.363s | 0m0.758s |
| 3 | 0m2.503s | 0m0.710s |
| 4 | 0m2.377s | 0m0.735s |
| 5 | 0m2.374s | 0m0.751s |

nextdate程序Block4相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 0m1.929s | 0m0.444s |
| 2 | 0m1.895s | 0m0.450s |
| 3 | 0m1.911s | 0m0.438s |
| 4 | 0m1.924s | 0m0.438s |
| 5 | 0m1.871s | 0m0.434s |

nextdate程序Block5相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 0m9.999s | 0m2.662s |
| 2 | 0m9.612s | 0m2.536s |
| 3 | 0m9.053s | 0m2.603s |
| 4 | 0m8.925s | 0m2.366s |
| 5 | 0m9.013s | 0m2.514s |

nextdate程序Block6相关变异体变异测试执行时间

|  |  |  |
| --- | --- | --- |
| 实验次数 | 传统变异测试方法 | 基于并发机制的变异测试优化技术 |
| 1 | 0m0.169s | 0m0.113s |
| 2 | 0m0.198s | 0m0.160s |
| 3 | 0m0.200s | 0m0.140s |
| 4 | 0m0.204s | 0m0.142s |
| 5 | 0m0.228s | 0m0.148s |

# 在 学 取 得 成 果

1. 在学期间所获的奖励

|  |  |  |
| --- | --- | --- |
| 奖励名称 | 授奖机构 | 授奖时间 |
| 人民三等奖学金 | 北京科技大学 | 2012.11 |
| 人民二等奖学金 | 北京科技大学 | 2013.11 |
| 优秀三好学生 | 北京科技大学 | 2013.11 |
| 第十一届“北京科技大学科技园杯” 学生创业计划大赛优胜奖 | 北京科技大学 | 2014.04 |
| 人民特等奖学金 | 北京科技大学 | 2014.11 |
| 优秀三好学生 | 北京科技大学 | 2014.11 |
| 2014年SRTP项目获市级一等奖 | 北京科技大学 | 2014.12 |

# 

# 致　　谢

在此论文最终完成之际，首先要向我的指导老师孙昌爱老师表达由衷的感谢，感谢他在我研究过程中对我的悉心指导。每周一次例会，使我能够定期向导师反馈毕设成果及遇到的困难，毕设过程中提交的相关文档老师都亲自审阅，批注修改。本次论文从选题到完成，遇到了很多困难，在老师的耐心指导下，都得以解决，使我能够按时完成本科毕设。孙老师严谨的治学态度、求实的科研精神和精益求精的工作作风让我备受激励，再次，我谨向孙老师致以诚挚的感谢和崇高的敬意。

还要感谢实验室师兄师姐们的帮助与支持，感谢他们在百忙之中，抽出时间，悉心地为我答疑解惑。尤其是薛飞飞师姐，从毕设开始就给予我极大的帮助，并且鼓励我提出自己的观点，见解，无论是在理论知识、实验设计、工具设计还是关于各个报告的撰写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，顺利完成毕业设计及论文。

感谢大学四年里传授我专业知识的所有老师以及父母对我的支持与鼓励，使我能够拥有四年充实而有意义的大学生涯。最后，谨以此文献给我的父母、老师、朋友和所有关心我帮助过我的人，谢谢你们。