变异测试优化技术综述

章华鹏　孙立帆 王子搏

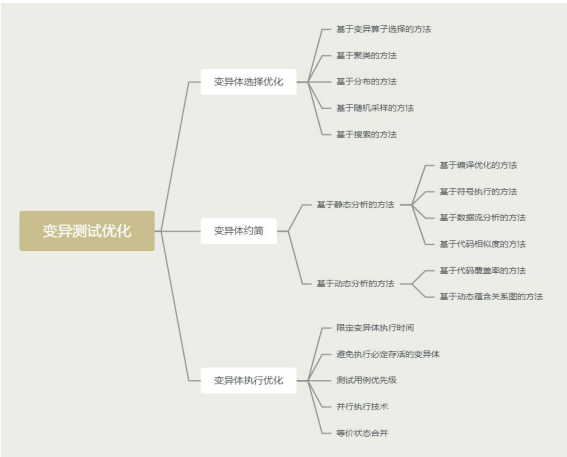
(南京大学软件学院 南京)

1 引言

变异测试在过去的四十年来受到广泛的研究。一方面，实证研究表明，与传统的基于数据流和控制流的测试方法相比，它拥有更加强大的能力，此外，它还可以用来作为软件工程研究领域的一个评估方法；但是从另一方面来说，由于通常需要执行和分析大量变异体，变异测试的开销是十分昂贵的。

因此，尽管变异测试十分强大，具有重大应用价值，但是通常实际应用中规模的程序会产生大量，规模较大的变异体，这使得变异测试在实际应用场景中受到阻碍。变异测试优化，主要是针对减少变异测试开销的优化，因此成为一个重要的研究领域，研究人员在近十年来提出了许多方法。

本文旨在对近十年来的变异测试优化技术做综述，通过对已有文献的整理和分析，我们将变异测试优化技术分为三个模块：变异体选择优化，等价/冗余变异体识别、变异体执行优化，并提出下图所示的研究框架。



本文的剩余部分按照如下方式组织：第二节介绍了关于变异测试的背景知识；第三节介绍了关于本篇综述的研究协议的相关内容，包括目标，文献选择和分析的标准等；第四节在引言中提出的研究框架的基础上对已有的技术方法做了进一步分类和分析；第五节简述了关于未来研究方向的展望；第六节是结束语。

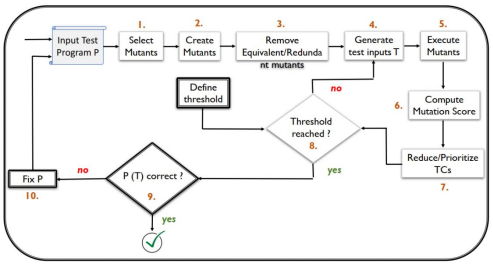
2 基本原理

让变异测试生成代表被测程序所有可能缺陷的变异体的策略并不可行，传统变异测试一般通过生成与原有程序差异极小的变异体来充分模拟被测软件的所有可能缺陷。其可行性基于两个重要的假设：老练程序员假设和缺陷可叠加假设。

老练程序员假设：一个老练程序员编写的错误程序与正确程序相差不大。

缺陷可叠加假设：复杂变异体可以通过耦合简单变异体得到，能够杀死简单变异体的测试用例可以杀死复杂变异体。

变异测试过程如图所示：



给定被测程序 p 和测试用例集 T ，首先根据

被测程序特征设定一系列变异算子；随后通过在原有程序 p 上执行变异算子生成大量变异体；接着从大量变异体中识别出等价变异体；然后在剩余的非等价变异体上执行测试用例集 T 中的测试用例，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。

3 研究协议

3.1 目标和研究问题

本文的目标是对近十年来关于变异测试优化，主要是关于减少变异测试开销的变异测试优化技术做归类和梳理，我们提出了如下的研究问题：

研究问题 1：目前关于减少开销的变异试优化技术主要有哪些?

研究问题 2：现有变异测试优化技术的效果如何？

研究问题 3：现有的变异测试优化技术是如何权衡开销减少和精度损失的？

3.2 文献查找、选择归类过程

第一阶段：粗略查找

这个阶段主要使用谷歌学术进行文献的查找，组员根据研究主题进行了分工，分三个模块查找：变异体选择优化，等价/冗余变异体检测，变异体执行优化。

在谷歌学术搜索引擎上通过主题关键词进行搜索，使用的关键词有如下：mutation testing, cost reduction, mutant selection, mutant operator, equivalent mutant, redundant mutant, mutant execution.

第二阶段：文献选择

搜集了文献之后，我们先通过文献来源的影响力进行了初步筛选，主要保留了来自 CCF 推荐会议期刊及一些相关的 Workshop 的文章。接着通过阅读文章的一些部分，如摘要，结论来确定是否符合主题，进行进一步的筛选。

第三阶段：文献阅读和归类

这一阶段每个人根据分工进一步更详细地阅读文献，并对文献中的研究方法做归类。

4 变异测试优化技术

4.1 变异体选择优化

变异体选择优化策略主要关注如何从生成的大量变异体中选择出典型变异体。据我们文献搜集结果，当下变异选择优化技术主要可以分为如下几类：基于变异算子选择的方法，基于聚类的方法，基于搜索的方法，基于分布的方法，基于随机采样的方法。

4.1.1 基于变异算子选择的优化

基于变异算子选择的优化从变异算子选择的角度触发，希望在不影响变异评分的前提下，通过对变异算子进行约减来大规模缩小变异体数量，从而减小变异测试与分析开销，最早由 Mathur 在 1991 年提出并被称作选择性变异（selective mutation），是近年来研究人员关注较多的方法，按照选择方法分类我们能够将其分为类级变异算子选择与传统变异算子选择；按照应用场景分类我们可以将其分为应用于安卓测试的变异算子选择优化与应用于 web 的变异算子选择优化。

（1）基于传统变异算子选择优化

据 Offutt 的分类，传统变异算子基于语法功能的差异可以分为三类：操作符替换，表达式修改，语句修改，共有 22 种变异算子，在 Yunqi Du 等人的工作中[9]，其将变异算子约减为 AOIU，AOIS，ROR，AORS，AORB 等 6 类表达式修改变异算子，在约减了变异算子减少了变异体数目的基础上，其变异得分与不约减下得到的基本一致，相较于其他工具（Randoop,Evosuit），使用了这 6 类的 MuGA 的代码覆盖率与变异得分均较高，是一种较好的变异算子优化方式。

传统变异算子的筛选也可以结合工具进行的，例如 Java 中的 PIT 工具[8]是通过操作字节码，只运行有机会杀死所使用的突变体的测试（即测试它执行突变体所在位置的指令），其十分健壮并与开发工具良好集成，而 PIT 从之前的有限操作符集拓展到当前的扩展的突变操作符集，这个过程进行了一次传统变异算子的生成与选择，以拓展的操作符来作为变异算子之后再通过修改操作符等方式来生成变异体进行测试，其对执行时间影响不大但是有效地提升了突变过程的有效性，在完成对变异算子的约减后，变异评分在 joda-time，commons-lang 中分别能够达到 85%，86%等。

（2）类级变异算子的选择优化

相较于传统变异算子，类级变异算子能够产生更少的变异体来得到与之相当的变异得分。不像传统变异算子如 MBC，MNC，SVR[4]，类级变异算

子是在类层级上定义的变异算子并按照特性可以分为继承，多态与动态绑定，方法重载，异常处理，混合方式这六类，不同特性的变异算子也可以进行进一步细分。而在 MuCpp 的相关工作[3]中就是使用类级变异算子来做 C++程序进行变异测试,其首先对 Clang 生成的 AST 语法数进行遍历，判断处变异算子能够插入的位置与种类；通过 git 不同分支存储不同的变异体并执行；MuCpp 通过记录每一个变异算子插入位置来避免执行重复变异体，同时定义变异算子生成规则避免生成 unproductive mutants 以减少最终变异体生成数目。而相较于传统的变异算子，使用 MuCpp 变异体的生成数量减少 46.6%，在 TCL Pro、XmlRpc++、Tinyxml2 上的总体变异得分分别达到 0.78，0.59，0.80 可见其虽然相较于传统变异算子的选择能够减少变异体数量，但是在变异得分方面传统变异算子选择优化仍然有其优势，二者互为补充。

（3）应用于安卓测试

按照变异测试应用场景分类，可以分为安卓测试与 Web 测试的应用，而这类的变异算子的选择方式有所不同。

在 jeff Offutt 等人的研究中[5]，其将应用于安卓测试的变异算子生成的过程是根据安卓应用程序的应用特征来定义变异算子，例如广泛使用 XML 文件来指定布局和行为，固定事件驱动方式，独特的程序生命周期结果，并且相较于使用传统变异算子对安卓程序进行变异体生成，使用特定的安卓变异算子来生成变异体，最终的变异得分为 0.632(传统的为 0.614)，生成变异体数量是传统的 152%。

而 Rafael A.P. Oliveira 等人从 GUI 界面入手[6]，其通过分析得出传统突变算子对于 GUI 级突变不够精确，因此定义了基于 GUI 的变异算子(可以自动生成，但是会收到 GUI 部件之间的依赖关系的约束)为 GUI 级变异体生成提供了全面支持。相较于传统变异算子应用于 GUI 界面时整体 11.27%的执行效率，基于 GUI 的变异算子达到了 83.9%，可见对于复杂的 GUI 图形界面，传统变异算子的测试并不充分，引入 GUI 变异算子能够较大程度提高测试效率进行测试优化。

（4）应用于 web

不同于应用于安卓应用的变异算子，应用于 web 的变异算子重点关注 JavaScript 预言并需要设置特定于 web 应用的变异算子 Rafael A.P. Oliveira 等人在 MUTANDIS 工具中实现了该类型的变异算子[7]：在 For，Function，DOM，JQuery，XHR 等部分定义了一系列变异算子，通过如将 undefined 替换为 null 等系列方式生成程序变异体之后再进行执行。在规模为 83 和 644 的测试集上进行运行，变异得分分别达到 67.8%、90.6%。

4.1.2 基于聚类的变异体选择优化

基于聚类的变异体选择优化技术是一种使用聚类算法如 K-Means 来选择变异体子集的方法。同一个集群中的每个变异体都可能被相同的一组测试用例杀死。因此，在划分聚类后，我们通过选择每个聚类中的一个或者部分变异体用于变异测试从而减少测试数量降低变异测试成本。

在 Yu-Seung Maa, Sang-Woon Kim 的研究[1]中，其在生成和执行变异体的过程中加入了一个过滤阶段，通过使用 AOR 突变操作符的方法来生成新的序列变异体，序列突变体是一个专门的程序来进行弱突变即对所有突变体只执行一次操作；然后执行部分序列变异体与弱变异体，计算其突变 ID 与突变值对，再以突变值对为标准互相比较，如果值相同，则突变 ID 给出的突变体为 c-overclap，归为同一个聚类；最后再按照聚类分别执行。

Xiangying Dang et al. [2]是按照变异体杀死难度与变异体相似性来进行聚类划分从而进行选择约减，将变异体按照杀死难度进行排序，比较变异体之间的相似性对其进行模糊聚类。

聚类方法基于一定原则对变异体进行分类，并从分类中选取代表加以执行以减少变异体执行数量从而降低成本。

4.1.3 基于随机的变异体选择优化

变异算子选择和变异体随机采样是变异体选择优化中两种常用的技术，前者通过限制把变异算子限制到一个子集上来减少生成的变异体的数量，后者通过在生成的变异体中随机采样来减少需要执行的变异体的数量，但是两者结合使用的可行性与效果是一个尚未成熟探索的领域。

L. Zhang et al. 把变异算子选择和随机采样选择结合起来[20]，提出了基于算子的随机变异体选择方法。

他们提出了八种基于变异算子选择之上的的随机采样策略，并在 11 个真实场景的 Java 程序上开展了实证研究，实验结果表明这种混合方法大幅度降低了变异测试的开销，通过采样变异算子选择后生成的变异体的 5%，将时间开销降到了原来的 6.54%,同时仍然保持可接受的变异得分计算结

果。

基于随机采样的变异体选择方法极大降低了变异测试的开销，且实现简单，但是其对变异测试得分计算的准确度的影响还有待更多研究验证。

4.1.4 基于分布的变异体选择优化

现有的很多变异体选择方法往往忽视程序的上下文，在平均效果上并未比随机选择方法要优越，难以消除许多无效或者冗余的变异体。

René Just et al. 首先提出了关于变异体效用(utility, i.e. usefulness)的假说[21]，他从等价性、平凡性、支配性(equivalence，triviality，dominance)三个角度来评估变异体的效用，并认为变异体的效用可以由程序的上下文推断出来，且我们应该为每一种变异算子，而不是变异算子组考虑上下文，当我们可以评估变异体的效用，就可以选出那些效用高的变异体子集。

基于如上观点，他们进而提出了上下文敏感的变异体选择方法，具体来说，通过从程序的抽象语法树(Abstract Syntax Tree)抽取的信息来建模程序上下文，基于分布选取更加分散的变异体。作者使用 129 种变异算子上开展了实验，实验结果表明程序上下文信息显著提高了对变异体效用评估的准确率。

未来还可以对这种建模方式进行进一步的拓展，探索与机器学习技术结合的可能性。

4.1.5 基于搜索的变异体选择优化

搜索是计算机科学和软件工程领域常用的方法，用来解决困难的优化问题，但是目前搜索方法在变异体选择领域的应用还没有像其在别的领域应用的如此广泛，因此一些研究者考虑将搜索方法引入变异体选择领域。

Pedro Delgado-Pérez et al. 将基于搜索的方法引入了变异体选择优化领域[22]，利用遗传算法来进行变异体的选择，以减少变异测试的开销。

他们基于提出的方法进行了一系列的实验，实验结果表明该方法选择出的一小部分变异体，也足以较为准确地计算变异得分，进而说明以遗传算法为代表的搜索算法在变异体选择领域应用的可行性。

4.2 等价/冗余变异体检测

源程序经过变异操作之后会得到大量的变异体，执行所有这些变异体会消耗大量时间和资源。并且所有变异体中有相当一部分是冗余的，对测试引导和评估测试用例没有帮助的甚至会损害到测试用例评估的结果[19]。 因此，需要有将这些冗余的变异体移除的方法，一个直观的想法就是通过尽可能多地检测出这些冗余的变异体来达到移除它们的目的。

这些冗余的变异体可以分为两类，第一类是等价变异体，这些变异体和源程序具有相同的语义，应该全部去除，第二类是蕴含变异体(若变异体 m1 被杀死时 m2 也会被杀死，那么称 m1 蕴含 m2)，被其他变异体蕴含的变异体都需要去除。

然而，判断变异体和源程序的语义等价关系或者变异体之间的蕴含关系，是一个不可判定问题。因此，只能通过启发式的方法来得到尽可能好的结果.

近年来，研究者在等价变异体和蕴含变异体的检测技术上提出了许多技术和方法，根据是否需要执行变异体，可以将这些方法分为两类：静态检测方法和动态检测方法。

4.2.1 静态检测法：基于编译优化技术的方法

M. Papadakis et al. 将编译优化技术(Trivial Compiler Equivalence)引入等价变异体的检测 [10]。该方法的假设是源程序和等价变异体经过编译优化之后会得到相同的二进制目标代码。他们在 18 个 benchmark 上做了实验(使用的是 gcc)，结果表明 TCE 能够检测出 30% 的等价变异体。

M. Kintis et al. 在 TCE 的基础上进行了进一步的实验[11]，将范围从 C 语言扩展到了 C 语言和 Java 语言，使用 javac，soot，gcc，在 C 语言和 Java 语言的程序上分别得到了 30% 和 54% 的等价变异体识别率。

TCE 是简单的，可扩展的，这使得 TCE 能够被比较容易地整合到变异测试流程中并应用于一定规模的程序。

4.2.2 静态检测法：基于数据流分析的方法

数据流分析是一类重要的静态分析方法，应用于变异测试中，它通过使用程序的数据流相关信息来确定那种类型的变异体会被生成和分析，这种技术考虑哪些变异目标在被执行或引用到的时候更容易被杀死。

M. Kintis 和 N. Malevris 提出了基于数据流分析的等价变异体检测方法[12]，他们介绍了一组特殊的数据流模式，这些数据流模式通常揭露了程序中产生等价变异体的地方，对这些地方进行变异操作产生的变异体很可能是等价变异体。基于数据流

分析的方法检测出了实验中所用的数据集 control mutant set 中 70% 的等价变异体。

M. Kintis 和 N. Malevris 基于上述提出的数据流模式做了进一步的实证研究，验证了上述方法的跨语言特性，并开发了一个静态检测等价变异体框架：MEDIC[13]，MEDIC 能够在 125 s 内检测出 56% 的等价变异体(总共 1300 个变异体)，显示了这种基于数据流分析的检测方法的效率和有效性。同时，MEDIC 是基于 Java 语言来编写的，但是在 JavaScript 的变异体上也取得了可观的效果，显示了基于数据流分析的方法的跨语言能力。

4.2.3 静态检测法：基于符号执行的方法

符号执行是一种静态分析技术，最初在 1976 年由 King JC 在 ACM 上提出。即通过使用抽象的符号代替具体值来模拟程序的执行，当遇到分支语句时，它会探索每一个分支，将分支条件加入到相应的路径约束中，若约束可解，则说明该路径是可达的。符号执行的目的是在给定的时间内，尽可能的探索更多的路径。

D. Holling et al. 采用一种基于符号执行的方法 Nequivack[14] 来进行 “非等价” 的检查。该方法通过在变异体上执行符号输入来判断变异体是否可杀死(即判断 “非等价” 性)，具体说来，通过与源程序在符号输入上的执行结果比较来判断变异体是否被杀死，无法被杀死的变异体则被归类为 unknown. Nequivack 能够高效地在 6 分钟内正确将所有变异体分类 (大约 10000 个)。

B. Kurtz et al. 提出了基于静态符号执行的冗余变异体检测方法[17]。该方法通过静态符号执行来建立静态变异体蕴含关系图 (Mutant Subsumption Graph)，作为对蕴含关系图 (Mutant Subsumption Graph) 的近似，来判定变异体之间的蕴含关系，在静态 MSG 上入度为 0 的点代表的变异体就认为是非冗余变异体。文章的实验结果表明了基于静态分析技术建立静态 MSG 以作为对 MSG 的近似的可行性，但是需要在更大的规模的程序上的实验来验证这个方法是否能够应用于日常真实场景。

这一类基于静态符号执行的方法在有效性上取得了较好的效果，但是由于符号执行本身的限制，在可扩展性上表现不佳，是否能够应用于规模较大的软件还有待进一步的研究。

4.2.4 静态检测法：基于代码相似度的方法

M. Kintis 和 N. Malevris 引入了镜像变异体的概念[15]，用来表示那些对相似的代码片段进行变异操作得到的变异体。他们认为镜像变异体在等价性上表现出相近或者相同的行为，因此当它们中的某一个是等价变异体的时候，它的镜像变异体也很有可能是等价变异体。

作者在真实场景中的程序上进行了实验，来验证这一想法，实验囊括了方法内和方法间的变异体，实验结果表明镜像变异体确实表现出在等价性上的相似行为，这一特点可以用来进行等价变异体的识别。

4.2.5 动态检测法：基于动态程序分析的方法

目前关于等价/冗余变异体的动态检测的方法比较少，这些方法主要都是先收集程序的一些运行时信息，然后根据这些运行时信息来对变异体进行建模，在此基础上进行等价/冗余变异体的识别

M. A. Guimarães et al. 提出了基于动态程序分析的冗余变异体检测方法[18]。作者提出了一种建立动态 MSG 来作为对 MSG 的近似的方法：对一个变异目标(文中是一个语句或者表达式)进行所有可能的变异，然后生成一个测试用例集，这个测试用例集能够杀死所有的变异体，之后计算杀死矩阵，根据杀死矩阵建立近似 MSG，在得到的近似 MSG 上入度为 0 的点所代表的变异体就是非冗余变异体。

作者在 32 个变异目标的 2341 个变异体上进行了实验，实验结果表明文章提出的方法在 20 个变异目标上取得了 100% 的有效性，在 29 个变异目标上取得了 95% 的有效性，此外，该方法减少了 52.53% 的时间开销。

Schuler et al. 提出了基于覆盖率变化的动态等价变异体检测方法[16]。他们认为那些没有被杀死但是改变了程序状态的变异体更有可能是 killable 的(即 Non-equivalent)，而覆盖率是程序状态的一个重要因素，作者用覆盖率的变化来指示非等价变异体的发现，即，如果一个变异体改变了测试用例覆盖率，那么我们认为它很可能是非等价变异体。

作者在 7 个 Java 程序上产生的 140 个变异体进行了实验，实验结果表明如果一个变异体改变了覆盖率，那么它有 70% 的可能是非等价变异体。

基于动态程序分析的等价变异体识别技术在有效性上拥有较好的表现，但是由于要实际执行变异体，因此开销较大。

4.3 变异体执行优化

变异体执行是变异测试中开销最大的环节，这个环节会在变异体上执行所有测试用例，假设有 n

个变异体，m 个测试用例，那么最多需要执行 n x m 次程序，这使得变异测试过程面临可扩展性问题，阻碍了变异测试的应用。

为了解决这个问题，研究人员提出了许多执行阶段的优化方法。其中主流的优化策略有如下四种：改变用例执行顺序、避免执行必定存活的变异体、限定变异体的执行时间、并行执行变异体和等价状态合并。

4.3.1 基于改变用例执行顺序的方法

测试用例优先级(Test Case Prioritization)是回归测试中常用的方法。测试用例优先级技术基于特定优先级准则，对测试用例进行优先级排序以优化其执行次序，旨在最大化优先级目标，以解决测试预算不足以执行所有测试用例的问题。

借鉴回归测试的思想，L.Zhang et al. 认为对历史版本的软件系统应用变异测试的信息可以帮助减少之后的变异测试开销，基于这个想法，他们提出了基于历史版本信息的测试用例优先级(Regression Mutation Testing , ReMT)[24]。

ReMT 通过对变异目标语句的覆盖率以及变异测试应用于历史版本软件的信息，进行优先级排序。具体来说，ReMT 的测试用例优先级排序基于这样的想法：如果一个测试用例能够杀死历史版本中的某个变异体 m，那么通常它也能杀死这个变异体在新版本中的对应的变异体。

作者使用 6 个真实场景的应用(代码行数在 3900 行代码到 8.88 万行代码之间)的版本仓库开展了实验，实验结果显示 ReMT 显著减少了测试开销。

上文提到的基于版本历史信息的测试用例优先级方法需要版本信息，这带来了 “冷启动”问题，为了解决这一问题，L.Zhang et al. 又提出了不需要版本信息的测试用例优先级和约简技术(Faster Mutation Testing ,FaMT)[23] 。

传统的测试用例优先级技术基于如下想法：当一个变异体被杀死之后不必在这个变异体上再运行测试用例，当一个测试用例在源程序上也无法覆盖变异目标语句时不需要执行这个测试用例。

FaMT 基于传统测试用例优先级技术做了进一步改进，首先 FaMT 需要进行一次初始运行来收集运行时信息(如是否覆盖变异目标语句等)，此外 FaMT 还动态维护每个测试用例在执行当前变异体之前的历史信息，通过这两个信息在运行时动态维护优先级。

作者在 9 个中等规模的 Java 程序上进行了实验，对 FaMT 进行评估，实验结果表明 FaMT 相对于随机方法减少了 47.52% 的执行次数，且 FaMT 方法带来的运行开销是相对可忽略的。

测试用例优先级技术在提高变异测试评估效果方面也有应用，Samuel J. Kaufman et al. 提出的测试完整性推进概率 (TCAP) [27]，作为一种新的变异体效用度量方法，能根据变异引发缺陷的可能性来评估变异体的效用，识别冗余变异体以及等效变异体，从而提高测试的完整性。

4.3.2 基于避免执行必定存活的变异体的方法

许多目前的变异体执行优化方法都是被测程序无关的方法。有些研究者从新的角度提出了一些新的方法，与这些被测程序无关的方法不同的是，在给定被测程序的情况下，又开发了其他方法来进一步优化变异执行过程。这一类的最新技术根据运行时的动态信息过滤不必要的执行。

Annibale Panichella et al. 提出的基于状态感染的数据压缩技术[25]，就是利用了行覆盖率和状态感染等信息，进一步优化了变异体执行。

同样，René Just，et al.也利用了运行时收集的动态信息，通过在未突变的程序上执行期间，监控复合表达式中受感染的执行状态的传播，来检测测试等效突变，同时通过划分具有相同感染状态的变异体来减少测试执行数量，实现数据压缩，优化变异体执行时间[26]。

4.3.3 基于限定变异体执行时间的方法

在变异体执行中，有时会遇到无限循环或者运行时间过长的情况，为了处理这种情况，学术界提出了各种不同的方法，但是其中最简单也有一定实用性的是限定变异体的执行时间。

Macario Polo Usaola et al.[28] 在在它们的变异体执行优化方法中引入了变异体执行时间限制参数，极大地减小了识别无限循环的开销。

但是这种基于时间限制的方法对变异得分的影响还有待更多的研究。

4.3.4 基于并行执行技术的方法

并行执行技术是一种提高程序执行效率，减少时间开销的方法。具体来说，应用在变异测试领域，并行执行技术通过在多处理器上并行执行多个变异体，以达到减少变异分析所需要的时间开销的目的。

近年来，一些研究人员在变异体执行技术上进行了一些探索，Mateo et al. 使用不同的网络配置对

5 种并行执行算法进行了实验，他们认为并行执行技术可以和其他变异测试优化技术结合起来以达到更好的效果[30]；N. Li et al. 将云计算的技术引入变异体执行优化[31]，在当下云计算蓬勃发展的背景下提供了新的思路。

随着硬件的能力变得越来越强大，变异体并行执行技术未来或许还会迎来更多的方法。

4.3.5 基于等价状态合并的方法

对于一个源程序的不同变异体，对于同一个变异位置的两个变异体，它们的变异目标语句大概率不是等价的，但是执行完这条语句之后程序的状态可能是一样的，此时我们称这两个变异体的语句同余当前状态，可以用同一个进程来表示它们。

基于这个想法，B. Wang et al. [29][32]提出了基于同余等价状态的方法，在 split stream (在第一条变异语句之前用同一个进程表示源程序和变异体) 的基础上做出了进一步改进，得到了比传统的 split stream 方法速度上快 2.56 倍的效果。

这种基于等价状态的方法在减少变异体数目和减少时间开销上都拥有较好的表现，但目前仍然存在不支持多线程程序等限制，在未来还有待更多的研究。

5 未来展望

5.1 变异体选择优化

变异体选择优化技术方面有基于变异算子的选择优化与基于聚类算法的选择优化，此外还有变异体随机选择优化与高阶变异体优化法等优化方式。变异优化技术中选择优化是重要一环，其通过合理减少变异体数目来直接减小变异测试的成本，在未来，变异测试与工具的结合将是一个趋势，随着工具的不断迭代与成熟，我们可以期待未来变异选择方式多样化与效果接近程度下的成本降低方法的不断涌现，去寻找进一步的选择优化方式。

5.2 等价/冗余变异体检测

等价/冗余变异体的识别方法，根据是否需要执行变异体可以分为静态识别方法和动态识别方法，目前较多的工作都采用静态的检测方法，而静态分析方法由于其过近似的特性在精度上天生就有一些损失，未来可以期待更多执行开销较小的动态分析方法或者以动态分析方法来辅助优化目前的静态分析方法的不足，在保持可接受开销的情况下尽量提高精度。

5.3 变异体执行优化

目前的变异体执行技术大多基于程序分析技术以及传统测试优化技术。近年来随着机器学习和深度学习技术的发展，在软件测试领域涌现了许多数据驱动的方法，比如机器学习技术在测试用例优先级上的应用。未来将经过数据驱动方法改进后的技术迁移到变异测试执行优化来进行进一步的应用应该是一个可期待的方向。

6 结束语

变异测试是一个非常有效的引导测试和评估测试的技术，但是它的开销也非常大。这使得许多研究者致力于变异测试优化技术的研究以减少变异测试的开销。本文对变异测试的部分优化技术做了整理，尝试对目前一些先进的方法做分类和归纳。我们搜集整理了 50 余篇文献，并留下了我们认为比较有代表性和影响力的作品，我们发现在近十年的作品中：程序分析(静态、动态)、随机方法、搜索等软件工程领域常用的方法在变异测试优化中都有较多的应用。

作为一篇综述，本文由于时间和人力所限在代表性和全面性上仍然有很大欠缺，但我们希望本文能使得读者对变异测试优化技术的主要方法有初步的认知。

参 考 文 献

[1] Mutation testing cost reduction by clustering overlapped mutants The Journal of Systems and Software 115 (2016) 18–30

[2] Enhancement of Mutation Testing via Fuzzy Clustering and Multi-Population Genetic Algorithm. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 48, NO. 6, JUNE 2022 2141

[3] Pedro Delgado-Pérez* , Inmaculada Medina-Bulo, Francisco Palomo-Lozano,Antonio García-Domínguez, Juan José Domínguez-Jiménez Assessment of class mutation operators for C++ with the MuCPP mutation system. nformation and Software Technology 81 (2017) 169–184

[4] P. Delgado-Pérez, I. Medina-Bulo, J.J. Domínguez-Jiménez, A. García Domínguez, F. Palomo-Lozano, Class mutation operators for C++ object oriented systems, Annals Telecommunications - Annales des télécommunica tions 70 (3-4) (2015) 137–148, doi:10.1007/s12243-014-0445-4.

[5] Lin Deng* , Jeff Offutt, Paul Ammann, Nariman Mirzaei Mutation operators for testing Android apps

[6] Rafael A.P. Oliveira，Rafael A.P. Oliveira，Zebao Gao，Atif Memon Definition and Evaluation of Mutation Operators for

GUI-level Mutation Analysis

[7] Shabnam Mirshokraie Ali Mesbah Karthik Pattabiraman: Efficient JavaScript Mutation Testing. 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

[8] Henry Coles ,Thomas Laurent, Christopher Henard,Mike Papadakis,Anthony Ventresque, PIT: A Practical Mutation Testing Tool for Java (Demo)

[9] Yunqi Du，Ya Pan，Haiyang Ao，O.ALEX，Yong Fan：Automatic Test Case Generation and Optimization Based on Mutation Testing. 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)

[10]M. Papadakis, Y. Jia, M. Harman and Y. Le Traon, “Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique,” 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 936-946, doi: 10.1109/ICSE.2015.103.

[11]M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon and M. Harman, “Detecting Trivial Mutant Equivalences via Compiler Optimisations,” in IEEE Transactions on Software Engineering, vol. 44, no. 4, pp. 308-333, 1 April 2018, doi: 10.1109/TSE.2017.2684805.

[12]M. Kintis and N. Malevris, “Using Data Flow Patterns for Equivalent Mutant Detection,” 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, 2014, pp. 196-205, doi: 10.1109/ICSTW.2014.21.

[13]M. Kintis and N. Malevris, “MEDIC: A static analysis framework for equivalent mutant identification”, Information and Software Technology,Volume 68,2015,Pages 1-17,ISSN 0950-5849

[14]D. Holling, S. Banescu, M. Probst, A. Petrovska and A. Pretschner, “Nequivack: Assessing Mutation Score Confidence,” 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2016, pp. 152-161, doi: 10.1109/ICSTW.2016.29.

[15]M. Kintis and N. Malevris, “Identifying More Equivalent Mutants via Code Similarity,” 2013 20th Asia-Pacific Software Engineering Conference (APSEC), 2013, pp. 180-188, doi: 10.1109/APSEC.2013.34.

[16]Schuler, David & Zeller, Andreas. (2013). Covering and Uncovering Equivalent Mutants. Software Testing, Verification and Reliability. 23. 10.1002/stvr.1473.

[17]B. Kurtz, P. Ammann and J. Offutt, “Static analysis of mutant subsumption,” 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2015, pp. 1-10, doi: 10.1109/ICSTW.2015.7107454

[18]M. A. Guimarães, L. Fernandes, M. Ribeiro, M. d’Amorim and R. Gheyi, “Optimizing Mutation Testing by Discovering Dynamic Mutant Subsumption Relations,” 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 198-208, doi: 10.1109/ICST46399.2020.00029.

[19]Rene Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2012. Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis? In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12). IEEE Computer Society, USA, 720–725. https://doi.org/10.1109/ICST.2012.162

[20]L. Zhang, M. Gligoric, D. Marinov and S. Khurshid, "Operator-based and random mutant selection: Better together," 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 92-102, doi: 10.1109/ASE.2013.6693070.

[21]René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring mutant utility from program context. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017). Association for Computing Machinery, New York, NY, USA, 284–294. https://doi.org/10.1145/3092703.3092732

[22]Pedro Delgado-Pérez, Inmaculada Medina-Bulo,Search-based mutant selection for efficient test suite improvement: Evaluation and results,Information and Software Technology,Volume 104,2018,Pages 130-143,ISSN 0950-5849,https://doi.org/10.1016/j.infsof.2018.07.011.

[23] Zhang L, Marinov D, Khurshid S. Faster mutation testing inspired by test prioritization and reduction[C]//Proceedings of the 2013 International Symposium on Software Testing and Analysis. 2013: 235-245.

[24]L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proc. ISSTA*, pages 331–341, 2012.

[25]Zhu Q, Panichella A, Zaidman A. Speeding-up mutation testing via data compression and state infection[C]//2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2017: 103-109.

[26] René Just, Michael D. Ernst, Gordon Fraser. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States. ISSTA’14, July 21–25, 2014, San Jose, CA, USA.

[27]Samuel J. Kaufman, Bob Kurtz, Ryan Featherman, Paul Ammann, Justin Alvin, René Just. Prioritizing Mutants to Guide Mutation Testing. 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)

[28]Macario Polo Usaola, Pedro Reales Mateo. Mutant Execution Cost Reduction, Through MUSIC(Mutant Schema Improved with Extra Code). 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.

[29]Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster mutation analysis via equivalence modulo states. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017). Association for Computing

Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/3092703.3092714

[30]Mateo, P.R. and Usaola, M.P. (2013), Parallel mutation testing. Softw. Test. Verif. Reliab., 23: 315-350. https://doi.org/10.1002/stvr.1471

[31]N. Li, M. West, A. Escalona and V. H. S. Durelli, "Mutation testing in practice using Ruby," 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2015, pp. 1-6, doi: 10.1109/ICSTW.2015.7107453.

[32]B. Wang, S. Lu, Y. Xiong and F. Liu, "Faster Mutation Analysis with Fewer Processes and Smaller Overheads," 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 381-393, doi: 10.1109/ASE51524.2021.9678827.