变异测试优化技术综述

刘承杰　赵凝晖　刘晓旭　王骁

(南京大学软件学院，南京，210093)

摘　要　变异测试是一种基于故障的软件测试技术，广泛用来评估测试用例集的充分性与软件测试技术的有效性。尽管变异测试具有较强的故障检测能力，但由于数量庞大的变异体导致了计算开销大的问题，阻碍了变异测试在实践中的广泛应用。针对于变异测试开销过大的这个问题，国内外的很多学者对于变异测试优化这一领域进行了一系列的研究并取得了一些成果。通过对已有的工作进行总结，本文将变异测试优化技术分为变异体选择优化和变异体执行优化两个模块。在变异体选择优化模块中，将现有的优化方法分成随机选择法、聚类选择法、变异算子选择法、高阶变异优化法以及程序分析法五个方面，并进行分类总结；在变异体执行优化模块中，从变异体检测优化、变异体编译优化以及并行执行变异体三个角度总结分类现有研究成果。最后对变异测试优化的未来研究方向进行展望。

关键词　变异测试；变异测试优化；变异体选择；变异体执行；综述

A Survey on Optimizing Mutation Testing Methods

Liu Chengjie　Zhao Ninghui　Liu Xiaoxu　Wang Xiao

(Software Institute, Nanjing University, Nanjing, 210093)

Abstract　Mutation testing is a fault-based software testing technique，which is widely used to evaluate the adequacy of a given test suite or the fault detection effectiveness of a given software testing technique. Although mutation testing has a strong fault detection capability, the high computation cost incurred by a huge number of mutants prevents mutation testing from being widely adopted in practice. Directed against the problem that mutation testing has occasion to excessive cost, a series of research have been conducted on mutation testing optimization by scholars globally. Summarizing the existing work and achievements, we can divide mutation testing optimization technology into two modules in this paper, which are called mutation selection optimization and mutation execution optimization. In the module “mutation selection optimization”, the existing methods are divided into five aspects: random selection method, cluster selection method, mutation operator method, high-order mutation optimization method and program analysis method. On this basis, we will summarize by the classification. In the module “mutation execution optimization”, we sort and summarize existing research results from the perspective of mutation detection optimization, mutation compilation optimization and parallel execution mutation. At last, we will foresee and outlook the future research direction to mutation testing optimization.

Key words　mutation testing; optimize mutation testing; mutant selection; mutant runtime; survey

1　引言

变异测试是一种基于故障的软件测试方法[1]，在最近几十年的时间里得到国内外学者的广泛关注。作为一种软件测试技术，变异测试与传统的借助数据流或者控制流分析来检验测试充分性的技术不同，该测试技术从评估和改进测试用例集的角

度来验证和提高测试充分性。具体来说，测试人员首先根据被测程序特征设计变异算子，变异算子一般在符合语法前提下仅对被测程序作微小改动。然后对被测程序应用变异算子可生成大量变异体，在识别出等价变异体后，若已有测试用例不能杀除所有非等价变异体，则需要额外设计新的测试用例，并将当前测试用例添加到测试用例集中，以提高测试充分性。除了可用于测试用例集的充分性评估，变异测试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[2]。

变异测试的概念最早由 Richard Lipton 在 1971 年提出，之后在 1980 年出现了第一个变异测试工具，在随后的时间里，研究人员对该领域展开了深入的研究并发表了一系列的论文，同时也催生出一些变异测试工具例如 MuJava、Muclipse 和 Javalanche 等，对于主流编程语言，例如 Fortran、C 以及 Java 等均有相应的变异测试工具。

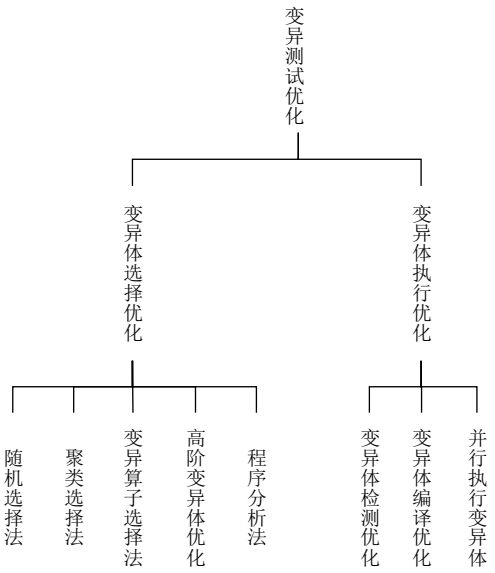

图 1　变异测试优化研究框架

虽然目前从学术界观点来看，变异测试是一种基于缺陷的成熟测试技术，但应用到工业界时却存在一个重要的技术难点，即变异测试分析过程中的计算开销较大，大量变异体的生成使得变异测试和分析时的开销极为高昂。其主要开销分布于变异体的生成、编译和执行过程中。为了将变异测试技术从学术界研究转化到工业界应用，需要提出有效的优化方法来减小计算开销。通过对已有文献的整理和分析，提出如图 1 所示的研究框架，该框架将变异测试的优化思路分为两类：变异体选择优化和变异体执行优化。同时将变异体选择优化的方法概括为五类：随机选择法、聚类选择法、变异算子选择法、高阶变异体优化法以及程序分析法，具体内容见第 3 章；将变异体执行优化的方法分为三类：变异体检测优化、变异体编译优化和并行执行变异体，具体内容见第 4 章。为了保证内容的连贯性和逻辑性，在介绍具体变异测试优化算法前，先介绍对变异测试的原理和有关术语定义做了相关介绍，具体内容见第 2 章。最后对全文进行总结，具体内容见第 5 章。

2　变异测试相关知识

2.1　基本假设

变异测试通过生成变异体，也就是程序中的微小变化并将其引入到程序代码来模拟被测试软件的所有可能缺陷。其可行性基于熟练程序员假设和耦合效应假设。

假设 1（熟练程序员假设）该假设由 DeMillo 等人在 1978 年首先提出，其内容是假设熟练的程序员因为编程经验比较丰富，编写出来的错误程序与正确的程序非常接近，只需要做少许修改就能成为正确的程序。该假设说明变异体能够模拟测试人员关注的最常出现的缺陷类型，同时基于这种模拟方式，能够在变异测试中暴露这些人工缺陷的测试套件一定能检测出待测程序中潜在的同类缺陷。

假设 2（耦合效应假设）该假设同样由 DeMillo 等人首先提出。他们认为，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到更为复杂的缺陷，这里简单缺陷指的是仅在原有程序上执行单一语法修改形成的缺陷，而复杂缺陷是指在原有程序上依次执行单一语法修改而形成的缺陷。在这个基础上，提出了简单变异体和复杂变异体的概念，也提出了变异耦合效应：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绝大多数的复杂变异体。

2.2　传统变异测试流程

传统变异测试流程如图 2 所示。

给定待测程序 p，根据待测程序和实际需要的类型生成变异体集合 $M=\{m_1, m_2, \ldots . m_k\}$，测试用例集 $T=\varnothing$，传统变异测试流程描述如下：

(1) 在原有程序 p 上执行变异算子生成大量变异体；

(2) 从大量变异体中识别出等价变异体
(3) 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
(4) 若能检测出所有的非等价变异体(或变异得分符合要求)则变异测试分析结束；否则需要添加新的测试用例到测试用例集 T 中，重复上述步骤。

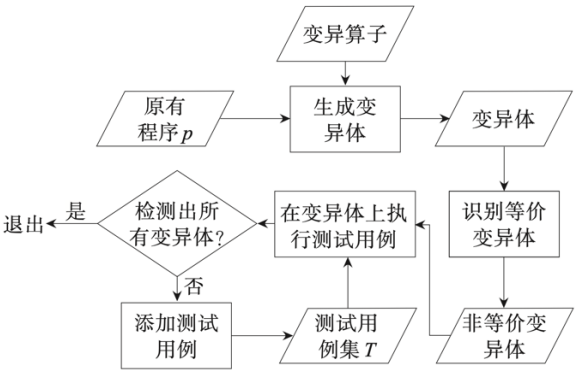

图 2 传统变异测试流程[2]

2.3 相关术语

下面对上述变异测试过程中的基本概念依次定义如下。

定义 1（变异算子）在符合语法规则的前提下，变异算子定义了一组语法转换规则，这种规则用于从原程序生成变异体。例如，对于原程序 $a+b>c$ 变异为 $a+b<c$，此时的变异算子即“>”变异为“<”。

在完成变异算子设计后，通过在原程序上执行变异算子可以生成大量变异体，根据执行变异算子的数目，可以将变异体分为一阶变异体和高阶变异体。

定义 2（一阶变异体）在原有程序 p 上执行单一变异算子并形成变异体 p'，则称 p'为 p 的一阶变异体。

定义 3（高阶变异体）在原有程序 p 上依次执行多次变异算子并形成变异体 p'，则称 p'为 p 的高阶变异体。若在 p 上依次执行 k 次变异算子并形成变异体 p'，则称 p'为 p 的 k 阶变异体。

表 1 一阶变异体与高阶变异体

程序 p	一阶变异体 p'	高阶变异体 p''
for(int i=0; i<8; i++)	for(int i=0; i!=8; i++)	for(int i=0; i!=8; i++)
sum += $a[i]$	sum += $a[i]$	sum −= $a[i]$

定义 4（可杀死变异体）若存在测试用例 $t(t\in T)$，在变异体 p'和原有程序 p 上的执行结果不一致，则称该变异体 p'相对于测试用例集 T 是可杀死变异体。

定义 5（可存活变异体）不存在任何测试用例 $t(t\in T)$，在变异体 p'和原有程序 p 上的执行结果不一致，则称该变异体 p'相对于测试用例集 T 是可存活变异体。

一部分可存活变异体通过设计新的测试用例可以转化成可杀死变异体，剩余的可存活变异体则可能是等价变异体。本文对等价变异体定义如下。

定义 6（等价变异体）若变异体 p'与原有程序 p 虽然在语法上存在差异，但在语义层面上与 p 相同，则称 p'是 p 的等价变异体。

表 2 等价变异体示例

程序 p	等价变异体 p'
for (int i = 0; i < 8; i++)	for(int i = 0; i != 8; i++)
sum += $a[i]$	sum += $a[i]$

由等价变异体的定义可以看出，这类变异体无法被任何的测试输入杀死，既不能有效地模拟故障缺陷，也无法改进测试数据充分性.因此在测试前，需要将程序中的等价变异体识别并移除。但等价变异体识别长期以来一直是研究领域和工业界所面临的一个主要难题，一方面，通过人工审查判断变异体的等价性的平均时间较长，另一方面，现有技术存在准确度低和拓展性差等问题，无法有效识别等价变异体。鉴于等价变异体检测是个不可判定问题[3]，本文在讨论变异测试优化技术时不讨论等价变异体检测问题。

定义 7（变异得分）变异测试分析最终通过输出变异评分来评估测试用例集的缺陷检测能力，变异得分 $MS(M, T)$可通过下式计算：

$$MS(M,T)=\frac{killed(M,T)}{|M|\text{-}eqv(M)}$$

其中，$killed(M, T)$表示测试用例集 T 杀死的变异体数量；$eqv(M)$函数表示等价变异体数量。由上式可以看出，变异得分的取值范围介于 0 到 1 之间，且其取值越高，代表测试用例集的实际缺陷检测能力越强。

从学术界观点来看，变异测试是一种基于缺陷的成熟测试技术，无论是 2.1 提到的相关假设还是 2.2 提到的变异测试流程，都证明了变异测试具有很强的可用性。但应用到工业界时却存在一个重要的技术难点，即变异测试分析过程中的计算开销较大，尤其体现在大量变异体的生成使得变异测试和分析时的开销极为高昂，且主要开销分布于变异体的生成、编译和执行过程中。为了将变异测试技术

从学术界研究转化到工业界应用，需要提出有效的优化方法来减小计算开销。下面将从变异体选择优化和变异体执行优化两个角度对最近的变异测试优化技术进行概述。

3 变异体选择优化技术

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，也就是在保证变异得分几乎不变的情况下，降低执行的变异体个数。该问题可以描述为：对于变异体集 M，测试用例集 T，寻找 M 的子集 M'使得 $MS(M, T)\approx MS(M', T)$。

已有的变异体选择优化方法可以简单分为 5 类：随机选择法、聚类选择法、变异算子选择法、高阶变异优化法以及程序分析法。

3.1 随机选择法

随机选择法的基本思想是从大量变异体中随机选择部分变异体。其基础是 Wong 和 Mathur 提出的单轮随机选择法[4]。具体来说，单轮随机法的思路是首先通过执行变异算子生成大量的变异体，然后根据定义的选择比例从生成的变异体中随机选择出该比例的变异体。其实验结果表明，若随机选择 10%的变异体，相对于选择所有变异体，变异评分仅减少 16%，且当选择比例超过 10%时，随机选择法是一种行之有效的优化办法。

在单轮随机选择的基础上，Zhang Lu 等人又提出了双轮随机选择法[5]。其核心思想是首先随机选择一种变异算子，然后从使用该变异算子生成的变异体中再随机选出一个变异体。Zhang Lu 等人选择西门子套件的 7 个 C 语言程序，对 Proteum 的 108 个变异算子进行选择。其实验结果表明，基于在 Proteum 的 108 个变异算子选择上，随机选择 7%的变异体就能达到与 Offutt 等人的采用变异算子选择法的实验的相似的稳定性和可行性。

上述方法都是从整个程序的角度进行分析，Ali Parsai 等人提出了类级的随机选择优化方法[6]。Ali Parsai 等人选择 12 个开源的 Java 项目进行实验。他们依据程序包含的类对变异体集合进行划分，分别对每个划分后的变异体集合随机选择并计算平均变异得分。其实验结果如图 3 所示。可以看到，当样本比例超过 10%时，这种随机选择的偏差基本维持在 1%以下。

在变异体选择的层面，Ali Parsai 等人采用皮尔逊相关系数（ρ）与肯德尔等级相关系数（τ_b）进行度量，这两个指标共同组成了“代表性”这个概念来评估变异体的被选样本集和整体集的覆盖率之间的关系。根据其实验结果，在保证 ρ 与 τ_b 在 0.75 以上时，在取得相同变异得分的情况下，进行相关性约束要比不进行约束的情况的变异体选择比例平均降低 18%。

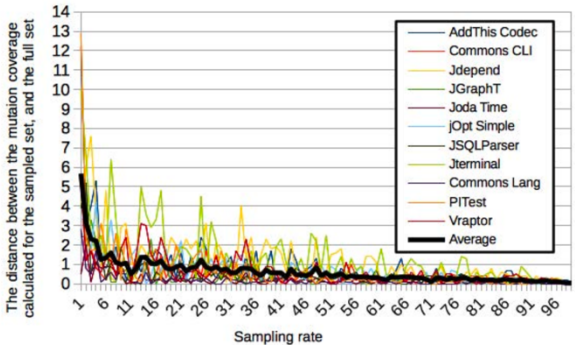

图 3　类级随机选择优化结果[6]

今年 R. Pitts 提出了支配变异体、准支配变异体以及非支配变异体的概念[7]，并发现当使用随机选择的变异体作为测试要求时，平均抽取支配变异体或准支配变异体的概率最初是非支配变异体的两倍；但随着测试的进行，选择等效变异体的概率会迅速压倒所有其他变异体类型。简而言之，支配变异体总是比非支配变异体更有可能被随机选择，但随着测试的进行，等价变异体很快成为最有可能被选择的。这些观察结果为随机选择的有效性以及等价变异所带来的问题的严重性提供了新的见解。

3.2 聚类选择法

聚类选择法采用聚类算法，如 K-means、Agglomerative Clustering 和粒子群算法等对变异体进行聚类分析而得名。具体来说，首先对被测程序 p 应用变异算子生成所有的一阶变异体；然后选择某一聚类算法根据测试用例的检测能力对所有变异体进行聚类分析，使得每个聚类内的变异体可以被相似测试用例检测到；最后从每个聚类中选择出典型变异体，而其他变异体则被丢弃，这样使得有代表性的变异体的结果取代了同一组中所有其他变异体的结果。

由于 K-means 算法对 K 值有很强的依赖性，聚类的效果容易被初始的聚类中心影响；Agglomerative cluste 算法中的簇如果结合了就不能被修改，这就限制了聚类的质量；以及传统的粒子群算法在数据量增大的时候迭代次数就会增加，同时收敛速度也会下降。王曙燕等人提出了基于改进

粒子群算法的变异体优化技术[8]，其核心思想是将 *K*-means 与粒子群优化算法相结合，利用 *K*-means 将粒子群中粒子群体划分为多个子群，然后使用粒子群优化算法，加大粒子更新对各自粒子所属区域的依赖程度，以此增强粒子群优化算法的全局寻优能力，加快算法的进化速度，提高收敛精度。

为了验证结果，王曙燕等人选择判断三角形形状程序 Triangle、循环队列操作程序 CircuQueue 和北卡莱罗纳州立大学研究团队在 Eclipse 平台开发的插件 Muclipse 附带的测试程序 Recipe，结果如图 4 所示。并与 *K*-means 算法、Agglomerative cluste 算法和粒子群算法做比较，结果都比较稳定且效果更好。

被测程序	优化前			优化后		
	变异体数目	测试用例数目	测试充分度	变异体数目	测试用例数目	测试充分度
Triangle	90	30	95.6%	33	12	95.6%
CircuQueue	114	34	93.9%	49	10	93.9%
Recipe	62	40	93.5%	21	9	93.1%

图 4　基于改进粒子群算法的聚类技术实验结果[8]

除了提出新的聚类技术，对于聚类范围应该扩大到什么程度的研究也是十分有意义的。Misun Yu 等人提出了一个变异聚类系统用于比较执行变异语句（块）后变异体的中间状态[9]。该实验通过在表达式、语句和块三个级别聚集变异体来比较将聚类范围扩大到什么程度能更好地降低成本。根据实验结果，与 9 个实际应用的弱活变异过滤方法相比，语句级别聚类减少了 24.44%的变异执行次数，表达水平聚类减少了 10.51%，块级聚类减少了 25.50%的执行次数。但是考虑到扩大比较范围所带来的额外成本，块级聚类带来的微小的改进是不实用的。基于这些结果，语句级变异聚类是三个聚类级别中最具成本效益的。

3.3　变异算子选择法

该方法从变异算子选择角度出发，希望在不影响变异评分的前提下，通过对变异算子进行约简来大规模缩小变异体数量，从而减小变异测试和分析开销。鉴于上文提到的 Zhang Lu 等人选择了西门子套件中的 7 个程序，在实证研究中对 2 种随机选择法和 3 种变异算子选择法进行比较[5]的结果表明，变异算子选择法相对于随机选择法来说并不存在明显优势，故本文对变异算子法不再做过多研究。

3.4　高阶变异优化法

高阶变异体优化法基于如下推测：执行一个 *k* 阶变异体相当于一次执行 *k* 个一阶变异体；高阶变异体中等价变异体的出现概率较小。

一般对于变异测试，尤其是高阶变异测试的关键问题之一是变异体数量的爆炸式增长，所以减少高阶变异体数量对于高阶变异测试而言非常重要。减少变异体数量的方法可以分为三类：减少变异算子的数量、选择有价值的高阶变异集以及减少变异的阶数。例如 Madeysk 等人采用临近算法、JudyDiffOp 算法选择变异算子[10]；Omar 等人采用遗传算法（GA）来构造有价值的、难以被杀死的高阶变异体[11]；Kintis 等人采用 1.EQM 算法考虑一阶和二阶变异体的执行行为，以剔除一阶等价变异体，即采用考虑二阶变异体的方式识别等价变异体[12]；Harman 等人采用一阶遗传算法（1.GA）[13]，Y. Jia,等人采用多目标优化之非支配排序遗传算法（NSGA-II）[14]以实现既减少变异体的数量，又对变异体的阶数进行约减。

3.5　程序分析法

该方法是通过使用程序分析领域中的有关算法对变异体执行的过程和结果进行分析，以实现冗余变异体的约减。例如 Kurtz B 等人采用定向增量符号执行（DiSE）技术创建静态变异体包含图（SMSG）[15]，其实现原理是符号执行（SE）描述在所有可能值上程序运行的结果，且 DiSE 专门用来分析给定程序两个版本间的差异，因此非常适合用做变异分析，同时 Kurtz B 等人也通过使用 MuJava 证明了静态分析的可行性。在此基础上，Guimarães M A 等人通过采用动态包含的方法，以当应用一组详尽的测试时，动态包含趋于稳定并接近真实包含的理论为基础，根据杀死矩阵来生成动态变异体包含图（DMSG），以确定最小变异集[16]。为了验证该方法的可行性，Kurtz B 等人在 MuJava 的基础上实现了一个新版本的工具 MuJava-M，然后，针对 5 个开源项目的 168 个类分别执行了 MuJava-M 工具，并在减少变异体数量、有效性和时间加速方面评估了该工具。结果表明 MuJava-M 执行变异分析的时间减少了 52.53%。

除了采用符号执行法，目前还有研究者使用数据流分析的方法来进行冗余变异体约减。孙昌爱等人提出一种基于数据流分析的冗余变异体识别方法。该方法的过程是大致如下：首先，分析待测程序 *p* 的程序结构，得到 *p* 的块规则文件；然后，逐个比较变异体程序和源程序，得到每个变异体的变异位置，根据块规则文件判断变异体的块类别；其

次，对 p 进行数据流分析，生成变量定义、变量使用、定义-使用链等数据流信息；最后，根据预定义的冗余变异体识别规则，结合变异体的块类别和程序数据流信息，识别出冗余变异体集合[17]。钱葭南等人则是针对程序中的顺序语句所产生的变异体，基于故障的可达-感染-传播模型，提出了使用区间抽象域来表示程序状态，通过区间运算判断变异体之间冗余关系的算法；针对程序中的条件语句，基于谓词故障层级，分别给出了面向简单谓词和复合谓词的冗余变异体选择算法[18]。

4 变异体执行优化技术

除了通过减少变异体数量来指导变异测试优化外，也有研究人员从优化变异体执行时间角度展开 研究。已有研究工作可以简要分为 3 类：变异体检测优化、变异体编译优化和并行执行变异体。

4.1 变异体检测优化

在变异体执行优化时，可以通过提高变异体检测效率来提高变异测试分析效率。测试用例检测变异体的方式可以分为 3 类：强变异检测、弱变异检测和固定变异检测。

定义 8（强变异检测）给定被测程序 p 和基于程序 p 生成的一个变异体 m，若测试用例 t 在程序 p 和变异体 m 上的输出结果不一致，则称测试用例 t 可以强变异检测到变异体 m。

定义 9（弱变异检测）假设被测程序 p 由 n 个程序实体构成 $S=\{s_1, s_2, ..., s_n\}$。通过对程序实体 s_m 执行变异算子生成变异体 m，若测试用例 t 在程序 p 和变异体 m 中的程序实体 s_m 上执行后的程序状态出现不一致，则称测试用例 t 可以弱变异检测到变异体 m。

测试用例采用弱变异检测方式时，并不需要执行完所有的程序实体，从而提高了测试用例的检测效率。但由于不同程序实体间存在依赖关系，若用于评估测试用例集充分性，弱变异检测方式要弱于强变异检测方式。因此弱变异检测方式通过牺牲变异评分的精确性来提高变异测试分析的效率。

为了进一步改进弱变异测试，Papadakis 和 Malevris 将变异前后的语句结合起来构造了一个变异分支，并将所有的变异分支插入到原程序的适当位置，形成了一个新的程序。覆盖变异分支的测试数据也会在弱变异测试中杀死相应的变异体。他们的实验表明，在变异分数没有太大损失的情况下，该方法进一步节省了执行时间[19]。然而，新程序中的大量变异分支极大地增加了复杂性。此外，Papadakis 和 Malevris 通过覆盖选定的路径生成基于变异的测试数据[20]。同样地，如果不进行优化约减，大量的变异体会导致很大的搜索空间，这会严重增加生成测试数据的成本。

为了解决这个问题，很多研究人员通过分析变异体的包含关系对变异体进行约减。例如 Gong D 等人提出了类似 MSG（变异包含图，详见 3.5）的优势关系图（dominance relation graph）方法以进行变异体约减[21]。该方法首先根据变异前后的语句构造变异分支，将所有变异分支融合到原程序中形成一个新的程序。然后，分析新程序中变异分支之间的优势关系。最后得到与构造前的变异体相对应的非支配变异体分支。但是该方法仅适用于减少传统（方法级）变异算子产生的变异体，不适用于类级算子；而且，对于该方法，需要手动分析源代码，这将导致可扩展性问题，实际应用性不高。

固定变异检测是介于强变异检测和弱变异检测之间，同时考虑了变异体执行的中间状态和最终输出结果的方法。该方法由 Woodward 和 Halewood 提出，随后 Jackson 和 Woodward 利用 Java 编程语言中的线程开发出一种并行固定变异检测方法[2]。

4.2 变异体编译优化

早期变异测试中变异体编译优化一般基于解释器，变异体采用解释执行方式输出执行结果。随后又提出基于编译器的优化技术，这类技术将每个变异体均编译为可执行程序，然后测试用例在可执行程序上直接运行。下面将总结近十年来的一些经典的变异体编译优化方法。

Hariri F 等人提出的 SRCIROR[22]是一个用于在 C/C++源代码（SRC）和 LLVM 编译器中间产物（IR）级别执行变异测试的工具集。在源代码级别，SRCIROR 通过对 Clang AST 执行模式匹配来识别程序结构，然后 SRCIROR 会将相关的变异算子应用于找到的结构，在 IR 级别，SRCIROR 使用 LLVM 通道找到应该改变的指令，然后直接改变这些 IR 指令。SRCIROR 能够处理 Clang 可以处理的任何程序，实现以最小的开销扩展到大型程序，同时对于一小部分无法编译的无效变异体。SRCIROR 可以在 SRC 和 IR 级别使用相同的变异算子类执行变异测试，并且 SRCIROR 很容易扩展以支持更多的算子。

Just R 等人实现的 MAJOR 是一种编译器集成

的非渗透测试工具，可为任意目的提供快速故障播传播，从而实现高效的变异分析[23]。与 Jumble、MuJave 以及 Javalanche 等原有工具相比，MAJOR 是集成到 Java 编译器中的，不需要特定的变异分析框架。因此，它可以在任何基于 Java 的环境中使用。Just R 等人还提出了一种可以集成到编译器中并提高变异分析效率的条件变异方法[24]。它基于转换 AST 以收集所有变异体以及生成的汇编代码中的原始程序。该方法的名字来源于对变异体的条件语句和表达式进行插桩的做法。

对于等价变异体检测，也可以通过编译优化的方式实现。Kintis M 等人提出简单编译器等效（TCE）这样一种简单、快速且广泛适用的检测等效变异体的技术[25]。其核心思想时认为具有相同机器代码的任何两个程序版本等效。TCE 简单地编译每个变异体，将其机器代码与原始程序的机器代码进行比较，还通过将每个变异体与位于同一单元（即功能）中的其他变异体进行比较来检冗余变异体。在此基础上，Delgado-Pérez P 等人在进化变异测试（EMT）上应用 TCE 来进行变异体约减[26]。对其应用可以分为两种情况，一种是在执行 EMT 之前应用 TCE，在这种情况下，编译所有突变体以创建可执行文件，TCE 将它们与原始程序的可执行文件进行比较。然后标记由 TCE 检测为等价的那些突变体以避免它们可以在 EMT 的执行期间使用。另一种是在执行 EMT 期间应用 TCE。在这种情况下，使用 TCE 分析在每一代中选择的突变体。那些被认定为等价的突变体通过分配给它们的适应值来惩罚。

4.3 并行执行变异体

除了上述编译优化技术，研究人员也尝试借助采用并行执行方式来进一步提高变异测试分析效率，该方法起源于 Mathur 和 Krauser 借助向量处理机提高变异测试分析效率[27]。近十年内的研究成果总结包括：Mateo P R 等人提出了 Bacterio 测试工具[28]，它允许测试人员在并行处理器之间分配突变体的执行，从而减少突变体的执行时间；N. Li 等人用企业级云计算技术亚马逊弹性云计算（Amazon EC2）来降低变异测试的计算成本[29]；Rahul Gopinath 等人提出了一个简单的源代码转换程序[30]，对于该程序每次遇到变异点都会导致执行分叉：父进程继续执行测试用例，并将该特定点标记为已发生变异；子进程用变异算子替换特定的变异点，并忽略所有其他变异点，实现各子进程并行执行以减少变异测试的时间；Absalom E. Ezugwu 提出了基于萤火虫元启发式算法的优化算法——FAII 算法[31]，旨在通过结合增强的全局最佳解决方案更新机制和基于自适应变异的局部和全局邻域搜索方案来提高标准萤火虫算法的性能，应用 FAII 算法能够从优化分布式计算环境中的任务和资源调度的角度来优化变异分析在分布式计算环境下的执行时间。

5 结束语

变异测试作为一种面向软件缺陷的测试技术，得到国内外研究人员的关注，并取得了大量研究成果。本文从变异测试优化角度对已有的研究工作进行了总结。虽然目前针对变异测试优化的研究已经取得了大量的成果，但该领域，即变异体的生成、编译和执行过程中仍存在很多研究点值得关注。虽然目前变异测试由于开销较大而在实践中并未得到广泛关注，但考虑到目前学术界对于变异测试优化技术研究成果的不断涌现，变异测试在工业界的应用依然有比较广阔的前景。

参 考 文 献

[1] DeMilo R A，Lipton R J，Sayward F G.Hints on test data selection:Help for the practicing programmer.IEEE Computer， 1978，11(4):34-41

[2] Chen Xiang, Gu Qing. Mutation Testing: Principal, Optimization and Application, Journal of Frontiers of Computer Science and Technology, 2012, 6(12): 1057-1075 (in Chinese)
(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.)

[3] Offutt J, Pan Jie. Automatically detecting equivalent mutants and infeasible paths[J]. Software Testing, Verification and Reliability, 1997, 7(3): 165-192.

[4] Mathur A P, Wong W E. An empirical comparison of data flow and mutation-based test adequacy criteria[J]. Software Testing, Verification and Reliability, 1994, 4(1): 9-31.

[5] 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 Soft- ware Engineering (ICSE 10). New York, NY, USA: ACM, 2010: 435-444.

[6] Parsai A, Murgia A, Demeyer S. Evaluating random mutant selection at class-level in projects with non-adequate test suites[C]//Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. 2016: 1-10.

[7] R. Pitts, "Random Mutant Selection and Equivalent Mutants Revisited," 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2022, pp. 170-178, doi: 10.1109/ICSTW55395.2022.00040.

[8] Wang Shuyan, Yang Yue, Sun Jiaze. Mutants selection based on improved particle swarm optimization algorithm[J]. Application Research of Computers, 2017, 34(3): 752-755
(王曙燕, 杨悦, 孙家泽. 基于改进粒子群算法的变异体选择优化[J]. 计算机应用研究, 2017, 34(3): 752-755)

[9] Yu M, Ma Y S. Possibility of cost reduction by mutant clustering according to the clustering scope[J]. Software Testing, Verification and Reliability, 2019, 29(1-2): e1692.

[10] L. Madeyski, W. Orzeszyna, R. Torkar, M. Józala, Overcoming the equivalent mutant problem: A systematic literature review and a comparative experi- ment of second order mutation, IEEE Trans. Softw. Eng.. 40 (1) (2014) 23–44.

[11] E. Omar, S. Ghosh, D. Whitely, Constructing subtle higher order mutants for Java and AspectJ programs, in: Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering, ISSRE, 2013, pp. 340–349.

[12] M. Kintis, M. Papadakis, N. Malevris, Isolating First Order Equivalent Mutants via Second Order Mutation, in: IEEE Fifth International Conference on Soft- ware Testing, Verification and Validation, 2012, pp. 701–710.

[13] M. Harman, Y. Jia, P. Reales Mateo, M. Polo, Angels and monsters: An empir- ical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE'14, ACM, New York, NY, USA, 2014, pp. 397–408.

[14] Y. Jia, F. Wu, M. Harman, J. Krinke, Genetic Improvement using Higher Order Mutation, in: GECCO Companion'15: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, 2015, pp. 803–804.

[15] Kurtz B, Ammann P, Offutt J. Static analysis of mutant subsumption[C]//2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2015: 1-10.

[16] Guimarães M A, Fernandes L, Ribeiro M, et al. Optimizing mutation testing by discovering dynamic mutant subsumption relations[C]//2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 2020: 198-208.

[17] Sun Changai et al. A Data Flow Analysis Based Redundant Mutants Identification Technique, CHINESE JOURNAL OF COMPUTERS[J], 2019, 1. (in Chinese)
(孙昌爱, 郭新玲, 张翔宇, 等. 一种基于数据流分析的冗余变异体识别方法[J]. 计算机学报, 2019, 1.)

[18] QIAN Gen-nan, WANG Ya-wen, GONG Yun-zhan, MENG Fan-rong. A Method for Finding Redundant Mutants in Mutation Testing[J]. Acta Electronica Sinica, 2017, 45(8): 1970-1975. (in Chinese)
(钱葭南, 王雅文, 宫云战, 孟凡荣. 一种变异测试中冗余变异体的寻找方法[J]. 电子学报, 2017, 45(8): 1970-1975.)

[19] M. Papadakis, N. Malevris, Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing, Softw. Qual. J. 19 (4) (2011) 691–723.

[20] M. Papadakis, N. Malevris, Mutation based test case generation via a path se- lection strategy, Inform. Softw. Technol. 54 (9) (2012). 915–312

[21] Gong D, Zhang G, Yao X, et al. Mutant reduction based on dominance relation for weak mutation testing[J]. Information and Software Technology, 2017, 81: 82-96.

[22] Hariri F, Shi A. Srciror: A toolset for mutation testing of c source code and llvm intermediate representation[C]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018: 860-863.

[23] Just R, Schweiggert F, Kapfhammer G M. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler[C]//2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011: 612-615.

[24] Just R, Kapfhammer G M, Schweiggert F. Using conditional mutation to increase the efficiency of mutation analysis[C]//Proceedings of the 6th International Workshop on Automation of Software Test. 2011: 50-56.

[25] Kintis M, Papadakis M, Jia Y, et al. Detecting trivial mutant equivalences via compiler optimisations[J]. IEEE Transactions on Software Engineering, 2017, 44(4): 308-333.

[26] Delgado-Pérez P, Medina-Bulo I, Merayo M G. Using evolutionary computation to improve mutation testing[C]//International Work-Conference on Artificial Neural Networks. Springer, Cham, 2017: 381-391.

[27] Mathur A P, Krauser E W. Mutant unification for improved vectorization[R]. Purdue University, 1988.

[28] Mateo P R, Usaola M P. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases[C]//2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, 2012: 646-649.

[29] N. Li, M. West, A. Escalona, and V. H. S. Durelli, "Mutation Testing in Practice Using Ruby," in Mutation'15, 2015.

[30] Gopinath R, Jensen C, Groce A. Topsy-Turvy: a smarter and faster parallelization of mutation analysis[C]//Proceedings of the 38th International Conference on Software Engineering Companion. 2016: 740-743.

[31] Ezugwu A E. Advanced discrete firefly algorithm with adaptive mutation-based neighborhood search for scheduling unrelated parallel machines with sequence-dependent setup times[J]. International Journal of Intelligent Systems, 2022, 37(8): 4612-4653.