CWMT：一种基于并发机制的弱变异测试加速技术

孙昌爱[1)]　曾国峰[1)]　张守峰[1)]　唐　锦[1)]　李　宁[2)]　张世永[2)]　陈　艳[2)]

[1)]（北京科技大学计算机与通信工程学院　北京　100083）

[2)]（华北计算技术研究所　北京　100083）

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，但由于变异体数量多与变异执行时间长导致了高昂的计算开销，限制了其在测试实践中的应用. 已有研究从两个维度研究如何降低变异测试的计算开销：(1) 变异体精简. 通过不同策略减少变异体的数量，往往减弱变异测试的故障检测能力；(2) 变异执行加速. 通过优化变异测试的执行过程，缩短变异执行的时间. 通过分析变异测试过程的特点，不难发现多个变异体之间存在大量重复执行的代码. 本文从缩短变异测试执行时间的角度出发，提出了并发弱变异测试，通过并发控制和程序合成相结合的手段优化变异体的执行过程，减少变异体的执行开销. 具体说来，并发弱变异测试融合了并发与弱变异两种变异执行的加速机制：并发机制通过共享某个程序块的不同变异体在变异位置之前的程序状态来缩短变异位置前的执行时间；弱变异机制通过比较源程序与变异体在变异位置之后的程序状态确定测试是否通过来缩短变异位置后的执行时间. 采用 12 个 C 程序以经验研究的方式评估了所提方法的有效性和优化效率，分析了影响优化效率的因素，并比较了所提方法与传统变异测试、弱变异测试、并发变异测试等 3 种基线技术的性能. 实验结果表明，本文提出的并发弱变异测试技术显著提升了变异测试的效率，即减少 90% 以上的编译时间和 70% 以上的执行时间.

关键词　软件测试；变异测试；弱变异测试；变异优化

中图法分类号　TP311　　**DOI 号**　10.11897/SP.J.1016.2023.01409

CWMT: A Concurrency Based Acceleration Technique for Weak Mutation Testing

SUN Chang-Ai[1)]　ZENG Guo-Feng[1)]　ZHANG Shou-Feng[1)]　TANG Jin[1)]
LI Ning[2)]　ZHANG Shi-Yong[2)]　CHEN Yan[2)]

[1)] (*School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing*　100083)

[2)] (*North China Institute of Computing Technology, Beijing*　100083)

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 and a long testing period prevents mutation testing from being widely adopted in practice. Existing research work on reducing the cost of mutation testing is mainly divided into two categories: (1) mutant reduction, which reduces the number of mutants through various strategies, and thus affects the fault detection capability of mutation testing; (2) acceleration of mutation execution, which shortens the time of mutation execution through optimizing the process of mutation testing. It can be observed that the bulk of codes in multiple mutants are repeatedly executed through an analysis of the mutation testing

收稿日期：2021-11-17；在线发布日期：2022-12-29. 本课题得到国家自然科学基金(61872039, 62272037)、航空科学基金(2016ZD74004)、中央高校基本科研业务费专项资金资助项目(FRF-GF-19-19B)、中国电子科技集团第十五研究所创新基金项目(19010203)资助. **孙昌爱**(通信作者)，博士，教授，博士生导师，中国计算机学会(CCF)高级会员，主要研究领域为软件测试、程序分析、服务计算、软件体系结构. E-mail: casun@ustb.edu.cn. **曾国峰**，博士研究生，主要研究方向为软件测试. **张守峰**，硕士研究生，主要研究方向为软件测试. **唐　锦**，硕士研究生，主要研究方向为软件测试. **李　宁**，硕士，高级工程师，主要研究方向为软件测试. **张世永**，学士，工程师，主要研究方向为软件测试. **陈　艳**，硕士，高级工程师，主要研究方向为信息系统.

《计算机学报》

process. In this paper, we explore the improvement of mutation testing in terms of reducing its execution time and accordingly propose a concurrent weak mutation testing approach, which provides an optimal mechanism to removal of redundant executions by using the combination of concurrent controls and program synthesis techniques, aiming at shortening the execution time of mutation testing. The proposed approach incorporates two mechanisms for accelerating mutation execution, namely the concurrent mechanism which reduces the execution time before the mutation point by sharing the program states of a set of mutants whose mutation points belong to the same program block, and the weak mutation mechanism which reduces the execution time after the mutation point by immediately comparing the program states of a mutant and the original program to decide whether a test passes or not. An empirical study has been conducted in which 12 C programs are used as subject program were used to evaluate the effectiveness and efficiency of the proposed approach and analyze the impact factors of performance optimization, and compare its performance with three baseline techniques, namely traditional mutation testing, weak mutation testing, and concurrent mutation testing. Experimental results show that compared with the baseline techniques, the proposed approach can significantly improve the efficiency of mutation testing, with an average reduction rate of more than 90% of compilation time and 70% of execution time, respectively.

Keywords software testing; mutation testing; weak mutation testing; mutation optimization

《计算机学报》

1 引 言

变异测试是一种基于故障的软件测试技术,通过植入模拟程序员易犯的不同类型的故障,对待测程序进行相对充分性测试[1-3]. 与此同时,变异测试也用作一种测试准则,广泛用于评估测试用例集的充分性与测试技术的有效性. 经验研究表明,变异测试具有较强的故障检测能力,且变异体比手动植入的故障更接近真实故障[4]. 然而,应用于大型程序时,由于测试用例与变异体数量庞大,变异测试的执行时间长,高昂的计算开销限制了变异测试在软件测试实践中的广泛应用[5].

近年来,在如何降低变异测试的测试代价方面出现了大量的研究成果[5],主要包含两个方面:(1) 变异体数量精简方法:从如何减少变异体数量的角度来降低测试代价,主要方法包括变异体随机选择[6]、选择性变异算子[7]、高阶变异测试[8]、变异体群束[9]等;(2) 加速变异执行方法:从如何缩短变异测试执行时间的角度来降低测试代价,主要方法有变异体检测优化[10]、变异体编译优化[11]、弱变异测试[12]、并行执行[13]、变异体排序[10]、预测性变异测试[14]等.

在前期研究工作中,我们提出了一种并发变异测试技术[15],利用变异体执行过程中变异代码块之前的程序状态相同的特点,采用并发机制对变异测试的执行过程进行加速,即主进程执行相同的代码片段,子进程并发执行不同变异代码片段. 通过分析并发变异测试的执行过程,我们进一步发现在变异代码块之后仍然有大量的相同执行过程. 在单进程情形下实现多个块变异的并发策略时,若在获取块变异执行后的程序状态立即进行测试结果判定,则可以进一步减少变异执行时间.

针对上述问题,本文探索并发执行与弱变异测试相结合以进一步加速变异执行,提出一种并发弱变异测试技术. 将变异体进行合成并加入并发控制机制使变异体共享相同的执行路径,通过判断变异语句对程序状态的影响来预测程序的输出变化,从而缩短变异测试的执行时间. 我们采用 12 个 C 程序以经验研究的方式评估了所提方法的有效性,并比较了相关的基准技术的优化效果.

本文第 2 节介绍变异测试原理、程序块相关概念;第 3 节介绍并发弱变异测试技术;第 4 节采用经验研究的方式对所提技术进行评估;第 5 节介绍相关的工作;最后总结全文.

2 背景介绍

介绍变异测试、弱变异测试、程序块和并发变异测试等相关概念.

2.1 变异测试

变异测试的基本思想是向待测程序中植入各种类型的故障[1]，产生大量的错误程序版本，使用测试用例检测这些包含故障的错误程序，评估测试用例集揭露特定类型故障的能力.产生的故障版本程序称为待测程序的"变异体"；模仿某类故障的操作称为"变异算子"，变异算子一般在符合语法前提下对待测程序作微小的语法改动.若变异体与待测程序在执行某个测试用例后输出了不同的结果，则称该变异体被"杀死"，即该变异体中植入的故障被检测出来；若所有测试用例都无法杀死变异体，则称该变异体"存活".若某个变异体与待测程序的语义相同，即对任何输入都产生相同的输出，则称该变异体为"等价变异体".传统变异测试流程如图1所示，过程描述如下[1]：

(1)给定待测程序 p，生成变异体集合 M，给定测试用例集 TS；

(2)用 TS 运行原程序 p，生成预期输出；

(3)用 TS 运行 M 中的所有变异体，并与预期输出相比.如果不同，则变异体被杀死；否则，变异体存活；

(4)计算当前测试用例集 TS 的变异得分；

(5)如果变异得分满足要求，则测试结束；否则，向测试用例集 TS 中添加一个测试用例；

(6)重复步骤(2)～(5)，直至满足要求.

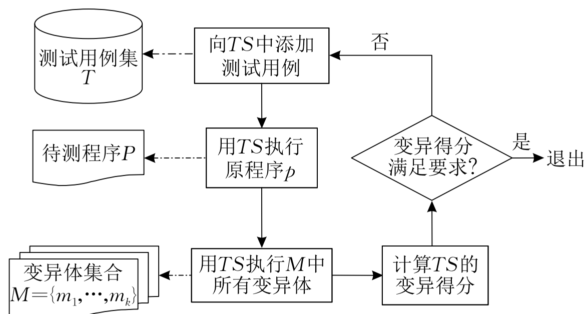

图1 传统变异测试流程

针对给定的测试用例集，能够杀死的变异体数量占所有非等价变异体数量的百分比，称为该测试用例集的"变异得分"(Mutation Score，MS).变异得分计算公式如下：

$$MS(p,TS)=\frac{N_k}{N_{all}-N_{eq}}\times 100\% \quad (1)$$

其中，p 表示被测程序，TS 表示测试用例集，N_k 表示被杀死的变异体数量，N_{all} 表示产生的变异体总数，N_{eq} 代表等价变异体的数量.

由式(1)可以看出，MS 的取值范围介于0到1之间，MS 取值越高，表明测试用例集的故障检测能力越强.变异得分直观地指明了测试用例集的故障检测能力，已经被广泛用来评估测试用例集的测试充分性，以及比较不同测试技术的有效性.近年来，变异测试已经应用到安全攸关程序(如航空、航天、交通等控制程序)的充分性评估，变异得分通常设定为100%，即测试用例集可以检测出所有的非等价变异体.

2.2 弱变异测试

在传统变异测试流程中，判断变异体被"杀死"必须同时满足如下三个条件：(1)可达性.变异点能够执行到；(2)必要性.变异点执行后程序状态发生改变；(3)充分性.状态改变能够传播到程序输出.采用上述方法的变异测试称为强变异测试[1].在强变异测试中，当一个变异体的输出与原程序的输出不同时，这个变异体认为被杀死.

当待测程序的规模增大时，数量庞大的变异体与测试用例将导致变异测试时间长，并引起巨大的计算开销.如果能够减少程序中不必要的执行，则能够大幅提升变异测试的效率.为此，Howden 提出了弱变异测试[12]，若同时满足可达性和必要性，则变异体被弱杀死.弱变异测试将程序中的基本计算结构定义为组件.假定一个原程序 p，c 是 p 中的一个组件，c' 是 c 的一个变异版本，p' 是包含 c' 的变异体.当一个测试用例 t 使 c' 能计算出与 c 不同的值时，则认为 p' 被杀死.相应地，Howden 定义了变量引用、变量赋值、算术表达式、关系表达式和布尔表达式五种程序组件类型.Offutt 与 Lee 在上述组件类型定义的基础上，提出了 EX-WEAK/1、ST-WEAK/1、BB-WEAK/1 和 BB-WEAK/N 四类弱变异[2].EX-WEAK/1 比较第一次执行包含被变异代码的最内层表达式后的程序状态；ST-WEAK/1 比较第一次执行被变异的程序语句后的程序状态；BB-WEAK/1 比较第一次执行包含被变异的程序语句的基本块后的程序状态；BB-WEAK/N 比较每次执行包含被变异程序语句的基本块后的程序状态.基本块是指具有单个入口点和单个出口点的程序语句的最长序列.对于修改赋值表达式右侧的变异算子生成的变异体，EX-WEAK/1 和 ST-WEAK/1 比较的结果是相同的.与 BB-WEAK/1 相比，BB-WEAK/N 能够解决循环语句中首次执行后的程序状态不会发生改变，而多次执行后程序状态发生改变的问题.

不同类型的弱变异中，受影响的状态部分也相应有所不同.对于 EX-WEAK/1，大多数变异仅影响

《计算机学报》

表达式的值；对于 ST-WEAK/1，比较赋值语句中被赋值的变量；对于 BB-WEAK/1 和 BB-WEAK/N，需要整个变量空间. 四种弱变异类型中，EX-WEAK/1 最容易实现，但是效果也最差；BB-WEAK/1 的效果与 ST-WEAK/1 很相近，且实现容易；BB-WEAK/N 的效果最好，但不易实现. 本文基于 BB-WEAK/N 实现弱变异测试与提出的并发弱变异测试.

2.3 程序块相关概念

本文在程序块层次上进行弱变异测试的加速执行问题，下面介绍程序块相关概念[16-18].

定义 1. 基本块. 一组顺序执行语句组成的区域，只有一个入口和出口. $BasicBlock=\{statement_{i,\cdots,j}\mid(i\le j)\wedge(\neg\exists n((n<i)\wedge(statement_n\prec statement_i)))\wedge(\neg\exists m((m>j)\wedge(statement_j\prec statement_m)))\}$，其中 $statement_i$ 表示程序中的语句行 i，$statement_{i,\cdots,j}$ 表示由语句 i 到 j 构成的语句段；$statement_x\prec statement_y$ 表示 $statement_x$ 执行后立即执行 $statement_y$.

定义 2. 选择块. 选择谓词表达式及其控制的语句行所组成的区域. $ChoiceBlock=(\varphi)\prec Region$，其中 φ 表示谓词表达式，$Region$ 表示基本块、选择块、循环块及其复合形成的一个控制依赖区域.

定义 3. 循环块. 当循环谓词表达式及其控制的语句行所组成的区域. $LoopBlock=((\varphi)\prec Region)^+$，其中 φ 表示谓词表达式，$Region$ 表示基本块、选择块、循环块及其组合形成的一个区域，表示执行一次或多次.

2.4 并发变异测试

变异测试依据待测程序生成大量变异体，并将测试用例集在原程序和变异体集上分别运行一次得到变异得分，过程如图 2 所示. 当变异体数量与测试用例的数量变大时，测试执行非常耗时. 通过分析变异测试过程的特点，不难发现同属某个程序块的不同变异体之间差异很小. 换言之，在一个测试用例执行的过程中，原程序和同属一个程序块的所有变异体在变异位置之前的程序状态是相同的. 基于上述观测，Sun 等人在强变异测试的基础上提出了一种并发变异测试技术[15]，使同属某个程序块的一组变异体在执行变异语句之前共享程序语句的执行结果，并生成子进程分别执行各个变异体的变异语句及之后的程序语句. 利用一次执行得到的变异语句前的程序状态，减少其它变异体同样程序片段的重复运行，在不增加额外计算资源的情形下有效缩短了变异测试执行时间.

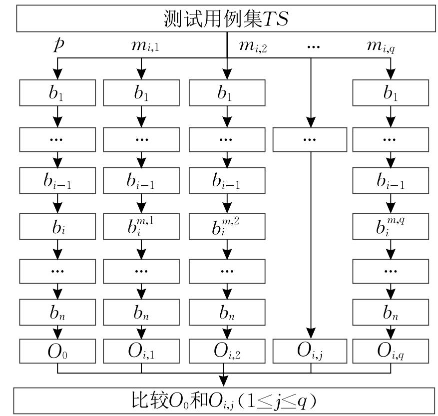

图 2 传统变异测试执行流程

图 3 示意了并发变异测试的基本原理. 假定一个原程序 p，包含程序块序列 $\{b_1,b_2,\cdots,b_n\}$，变异语句位于程序块 b_i 的一组变异体集合为 $\{m_{i,1},m_{i,2},\cdots,m_{i,q}\}$. 在程序块 b_i 执行前，p 的程序状态和每个变异体的程序语句都是相同的，因此没有必要重复执行原程序和每个变异体的程序块序列 $\{b_1,b_2,\cdots,b_{i-1}\}$. 不同于传统的变异测试，并发变异测试中在程序块 b_i 之前只执行一次程序块序列 $\{b_1,b_2,\cdots,b_{i-1}\}$. 具体说来，在执行包含变异语句的程序块 b_i 前，为每个变异体生成一个子进程. 这些子进程共享程序块 b_{i-1} 执行后的程序状态，并负责获取每个变异体在程序块 b_i 执行后的程序状态以及输出的结果，而原始进程则继续原程序的程序块 b_i 及其之后的程序块序列. 并发变异测试通过共享变异语句之前的程序状态来缩减测试执行时间，其优化效率与变异语句的位置相关.

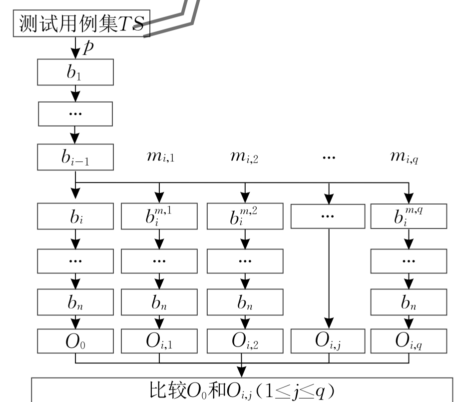

图 3 并发变异测试执行流程[15]

3 并发弱变异测试

介绍本文提出的并发弱变异测试技术，包括方法框架、关键技术（包括优化效率分析、程序中间状

《计算机学报》

态计算、程序合成与算法实现)和方法示例.

3.1　方法框架

在强变异测试中，不仅要求变异点是可达的且变异点的执行能够对程序状态带来改变，还要求程序状态的改变能够传播到程序的输出. 为了验证程序状态能否传播到程序的输出，从变异程序块到程序输出之间的程序片段仍然存在相似的执行. 因此，并发变异测试仅对变异程序块之前的执行过程进行了精简，但是变异程序块之后仍然存在大量的重复执行. 如果能够通过变异程序块执行后程序状态的比较结果对程序输出的执行结果进行预测，则可以进一步缩减变异执行时间. 相应地，我们将并发控制与弱变异测试相结合，提出了并发弱变异测试技术. 弱变异测试[2]通过在执行程序变异块后直接比较原程序和各个变异体的程序状态来判断变异体是否杀死，进一步减少程序变异块之后的代码执行. 加速弱变异测试的一个可行途径是引入多线程机制，即采用多线程执行不同的变异体. 该方法不仅对计算环境有特殊的要求，而且每个变异体需单独进行编译，并且无法避免变异语句块之前的程序片段的重复执行.

图 4 示意了并发弱变异测试的基本原理. 在程序块 b_i 执行前，程序块序列 $\{b_1, b_2, \cdots, b_{i-1}\}$ 仅执行一次；对于程序块 b_i，通过并发机制执行；程序块 b_i 执行后，直接比较原程序和各个变异体的程序状态. 若变异体中的块 b_i' 产生的状态和原程序中的块 b_i 产生的不同，则认为变异体的输出和原程序 p 的输出不同，变异体被杀死；否则，变异体存活. 通过程序变异块之前的程序状态共享和变异块之后的使用程序的中间状态判断变异体是否杀死，极大地精简了变异测试的整体流程.

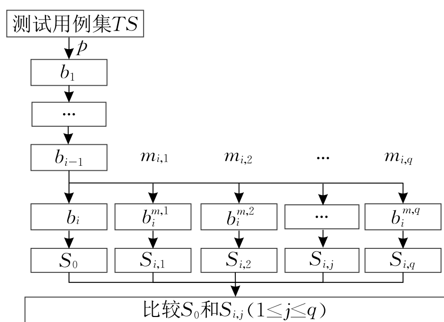

图 4　并发弱变异测试执行流程

进一步地，我们给出实施并发弱变异测试的框架，如图 5 所示. 首先，分析待测程序 p 的程序结构，根据程序块的定义对待测程序 p 进行分块，得到块规则文件(①)；然后，分析每个变异体中变异语句的位置，根据块规则文件，将变异体按变异位置所属的程序块进行分组，得到多组变异体(②)；对于每个程序块，提取该程序块对应代码段中的程序状态向量(③)；将每组中的全部变异体和待测程序 p 合成为一个并发块程序(④)；将测试用例集在并发块程序上执行，输出待测程序 p 和各个变异体的程序中间状态，判断变异体是否被杀死，统计测试结果并生成测试报告(⑤).

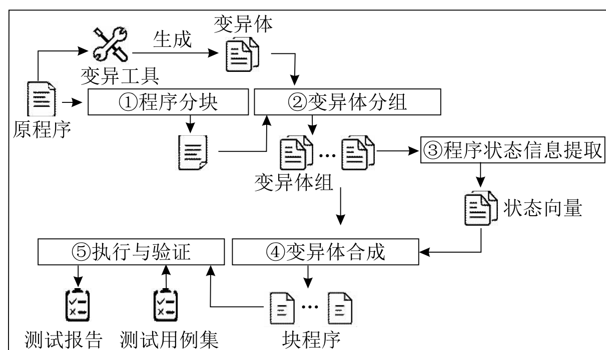

图 5　并发弱变异测试框架

3.2　优化效率分析

传统的变异测试与并发弱变异测试的执行过程具有很大的差异性. 具体说来，在传统变异测试的执行过程中，首先依据待测程序生成变异体，然后要对每个变异体进行编译获得可执行程序，再采用测试用例集中的测试用例运行每个可执行程序，直到测试用例全部执行完或变异体被杀死. 在并发弱变异测试的执行过程中，变异体通过分组与合成处理生成了块程序，需要对每个块程序进行编译，然后采用测试用例运行生成的可执行块程序，直到测试用例全部执行完或块程序中的包含的所有变异体都被杀死. 下面对并发弱变异测试在编译与执行两个阶段的优化效果进行理论分析.

(1) 并发弱变异测试在编译阶段的优化效率分析. 在并发弱变异测试中，按变异发生的位置和程序块结构将变异体分成不同的组，并将每组变异体合成为一个块程序，因此待编译的程序数量和变异体组的数量相关. 程序块的数量是小于程序语句的，而变异体的数量约等于程序语句数量与变异算子数量的乘积，因此并发弱变异测试中待编译的程序数量是远小于传统变异测试过程，其优化效率与被使用的变异算子数量成正比.

(2) 并发弱变异测试在执行阶段的优化效率分析. 假设待测程序共有 n 个程序块 $\{b_1, b_2, \cdots, b_n\}$,

《计算机学报》

变异语句位于 b_i 中，程序入口到 b_i 的执行时间为 eta，b_i 执行时间为 etb，b_i 之后的执行时间为 etc，变异体的数量为 q，且变异位置在同一程序块 b_i 中，测试用例的数量为 l. 在传统变异测试中，每个测试用例的执行覆盖原程序和每个变异体从输入到输出，消耗的时间为 $(eta+etb+etc)\times(q+1)\times l$. 在并发变异测试中，$b_i$ 之前的语句在同组变异体中只需要执行一次，消耗的时间缩短为 $eta'\times l+(etb+etc)\times(q+1)\times l$，其中 eta' 包括 b_i 执行前 fork 子进程的额外开销. 在弱变异测试中，执行过程在变异语句之后就停止了，消耗的时间缩短为 $(eta+etb')\times(q+1)\times l$，其中 etb' 包括 b_i 执行后程序中间状态保存的额外开销. 在并发弱变异测试中，对于每个块程序，在变异 b_i 之前的程序状态是共享的，只执行一次；在块 b_i 处，通过并发控制对变异块 b_i 分别执行，执行的次数为块程序包含的变异体数加上原程序的一次执行；在变异块 b_i 之后，直接输出程序的中间状态并结束程序；消耗的时间为 $eta'\times l+etb'\times(q+1)\times l$. 通常说来，$eta'$ 与 eta 相差较小. 为了便于分析，我们将 eta' 简化为 eta（即 $eta'\cong eta$），类似的，$etb'\cong etb$. 与传统变异测试相比，并发变异测试的时间减少了 $eta\times q\times l$，弱变异测试的时间则减少了 $etc\times(q+1)\times l$，而并发弱变异测试的时间减少了 $eta\times q\times l+etc\times(q+1)\times l$. 不难看出，并发变异测试在变异位置靠后时优化效率较高，弱变异测试在变异位置靠前时优化效率较高，而并发弱变异测试的优化效率不受变异位置所处程序结构位置的影响. 若变异位置靠前，并发弱变异测试能减少变异块之后的执行过程；若变异位置靠后，则能减少变异块之前的执行过程.

3.3 程序中间状态计算

在弱变异测试中，需要确定对比程序状态的位置，还需要定义如何表达一个程序的中间状态. 程序状态的表示与编程语言相关. Offutt 等人[2] 使用完整的变量空间和程序计数器等来表示 FORTRAN 程序的状态. Yu 等人[19] 使用类的属性值、方法的参数和局部变量来表示 Java 程序的状态. 本文实施并发弱变异测试时的 C 程序中间状态计算问题的解决方案如下.

首先，本文方法是基于程序块结构的，因此程序中间状态计算问题转化为如何确定与表示程序块执行后的程序中间状态. 程序中间状态在程序块执行前与原程序保持一致，而在不同变异程序块执行时可能发生改变，状态差异通过变量写入操作传播到程序块结束位置. C 语言中主要的写入操作有赋值、自增和自减. 本文通过分析程序块中相关操作来识别需要记录的变量，确定程序块执行后的程序中间状态.

其次，程序块中的函数调用也可能改变程序变量的状态. 由于追踪函数调用的代价较大，本文进行了简化处理. 若调用的函数与传递的参数相同，则变异体与原程序的程序状态不会因为函数调用而产生不同，反之则程序状态可能改变. 因此本文通过记录被调用的函数和函数参数来表示函数调用对程序状态的改变.

综上讨论，变异程序块执行后的程序中间状态表示为 $S=\langle V,C\rangle$，其中 V 代表程序块中被改变的变量集合，C 代表函数调用序列，C 中每个元素由被调函数的函数名和参数组成.

本文采用源代码插桩的方式实现对 S 的计算和保存. 对于 V 中的每个变量，在程序块后插入状态保存语句，即通过 C 库函数 fwrite 调用读取并保存变量的内存数据，当程序执行到该语句时将相应的变量状态保存下来. 例如，对于赋值语句“$x+=1$”中的变量 x，相应的状态保存语句为“fwrite(&x, sizeof(x), 1, stdout)”，其中，sizeof 在编译时获取变量的数据类型长度，stdout 标准输出在变异块执行前通过 C 库函数 freopen 重定向到不同文件. 对于函数调用而言，为了确定参数的数据类型长度，在被调函数的函数体开始处插入状态保存语句，记录函数名和参数值，当程序执行到该函数时将相应的程序状态保存下来. 通过比较原程序状态输出文件和变异体状态输出文件来判断变异体是否发生状态改变.

3.4 程序合成

并发弱变异测试将一组变异体与原程序合成为一个块程序，有效减少变异测试的执行开销，如图 6

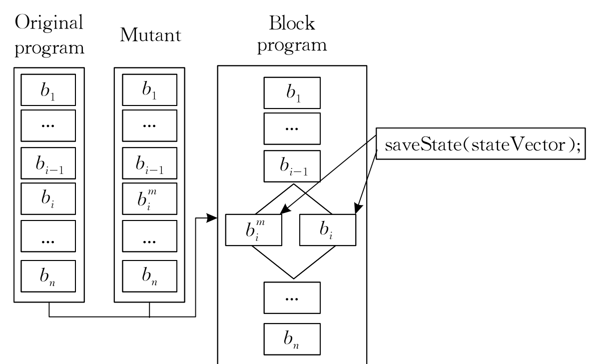

图 6 变异体合成示意图

所示.并发弱变异测试首先采用并发机制对变异程序块之前的程序片段进行合并处理，然后产生不同的子进程执行原程序和变异体中的变异程序块.与并发变异测试不同的是，在变异程序块执行后不再继续执行程序，而是输出程序的中间状态并分析测试结果.

在合成原程序和变异体时，需要对程序变异块进行处理.针对程序块的不同结构，本文设计了不同的合成规则.具体如下：

(1) 基本块合成规则.基本块中的每条语句都按顺序执行，只有一个执行出口.为获取程序变异块执行后的程序中间状态，只要在块的出口处获取程序的中间状态(saveState)并结束执行.

(2) 选择块合成规则.选择块是由谓词及其控制的代码区域构成，具有不同的分支.为了在不同的执行路径下都能获取程序的中间状态并结束执行，需要对两条执行路径都进行处理.图7示例了选择块的合成规则，其中注释语句指明了原程序中相应位置的代码.该程序片段的真分支为一个基本块，仅涉及变量 x 的重新赋值，程序中间状态需要获取 x 的值；类似的，假分支也是一个基本块，涉及变量 y 的重新赋值，相应的程序中间状态需要获取 y 的值.

```
if(x>y){
/* if(x<y){ */
      x+=1;
      saveState(&x);
}
else {
      y+=1;
      saveState(&y);
}
```

图7 选择块合成示例

(3) 循环块合成规则.循环块是由谓词及其控制的代码区域构成，可能执行多次.循环块的执行次数不确定，且在单次运行后变异造成的影响可能并不会造成程序状态发生变化，因此循环体中每次执行后的程序状态都应该保存，在循环块执行完后结束程序的执行.图8示例了循环块的合成规则，注释语句指明了原程序中相应位置的代码.该程序片段的循环体为一个基本块，仅涉及变量 x 的重新赋值，程序中间状态需要获取 x 的值.

```
int x=1;
for(int i=1; i<=100; i++){
/* for(int i=1; i<100; i++){ */
      x *=i;
      saveState(&x);
}
```

图8 循环块合成示例

3.5 算 法

在上述关键技术的基础上，我们进一步提出并发弱变异测试算法(见算法1).该算法的输入为待测程序 p、变异体集合 M、块结构信息 B 和测试用例集 TS，其中 m_i 指明了变异发生的位置 s_i，b_i 指明了程序块包含语句的行号.算法的输出是变异体集在执行测试后的存活状态 MSS，其中 $\langle m_i, s_i\rangle$ 指明了变异体 m_i 在执行后的存活情况 s_i，被杀死或存活.算法主要过程如下：(1) 首先进行初始化.对 MSS、变异体组集合 G 和块程序集合 BK 进行初始化；(2) 按变异位置对变异体进行分组.对于每个变异体，将每条语句与原程序相比，找到变异位置，根据变异位置和程序分块信息将变异体分组；(3) 将每组变异体进行合成.对于每个变异体组的所有变异体，依据3.3节讨论的程序中间状态计算方法获取该组对应程序块的程序中间状态向量；依据3.4节讨论的程序合成方法为每组变异体生成一个块程序，首先写入必要的控制代码，将变异块之前的程序代码写入块程序，然后在模块中写入该组中所有变异体在变异块中的程序语句，再写入对程序中间状态的处理和执行结束控制，最后向块程序中写入块 b_i 之后的语句，保证程序完整性；(4) 执行测试用例.对于每个块程序，执行测试用例集中的所有测试用例，比较原程序和所有变异体的程序中间状态.若不同，则变异体被杀死，将变异体的存活状态存入 MSS 中.当所有的变异体被杀死或所有的测试用例都执行后，测试执行完毕.

算法1. Algorithm of concurrent weak mutation testing.

INPUT: $P, M=\{m_1, m_2, \cdots, m_L\}, B=\{b_1, b_2, \cdots, b_V\}$,
$TS=\{t_1, t_2, \cdots, t_T\}$

OUTPUT: $MSS=\{\langle m_1, s_1\rangle, \langle m_2, s_2\rangle, \cdots, \langle m_n, s_n\rangle\}$

PROCEDURE:

1. INITIALIZE: MSS to an empty set, G to an empty set, BK to an empty set
2. FOR m_i in M DO
3.   FOR b_j in B DO
4.   IF notEqual($statement_{m_i, b_j}$, $statement_{p, b_j}$) DO
5. m_i belongs to g_j
6.   END IF
7. $G=G\cup\{g_j\}$
8.   END FOR
9. END FOR
10. FOR b_i in B DO

《计算机学报》

```
11.     extractStateVector(b_j)
12. END FOR
13. FOR g_i in G DO
14.     generate block program bk_i
15.     BK=BK∪{bk_i}
16. END FOR
17. FOR bk_i in BK DO
18.     FOR t_j in T DO
19.         IF isAllKilled() DO
20.             BREAK
21.         execute t_j on bk_i, produce test result of mutants
            RTM and test result of original program rt_p
22.         FOR rt_k in RTM DO
23.             IF notEqual(rt_k, rt_p) DO
24.                 MSS=MSS∪{⟨m_i, isKilled⟩}
25.             END IF
26.         END FOR
27.     END FOR
28. END FOR
29. RETURN MSS
```

算法复杂性分析. 假设变异体数量为 L，程序语句数量为 S，程序块数量为 V，测试用例集的数量为 T. 在分组过程中，对于每一个变异体，需要遍历语句寻找变异位置，其时间复杂度为 $O(S)$，对于变异体集合的时间复杂度为 $O(S\times L)$. 在合成过程中，对于每一组变异体，假设其变异体的数量为 n_i，其时间复杂度为 $O(S\times n_i)+O(1)$，总的时间复杂度为 $\sum_{i=1}^{V}(O(S\times n_i)+O(1))=O(S\times L)+O(V)$. 在测试过程中，对于每个块程序运行所有的测试用例，直到其中的所有的变异体被杀死或运行结束，其时间复杂度为 $O(T)$，总的时间复杂度为 $O(V\times T)$. 综上，并发弱变异测试算法总的时间复杂度为上述几项的和，即 $O(S)+O(S\times L)+O(V)+O(V\times T)=O(S\times L)+O(V\times T)$.

3.6　方法示例

采用 minmax 程序示例并发弱变异测试的应用. 图 9 示意了示例程序的代码和部分变异体. 方法使用过程如下：

(1) 代码规格化. 通过代码规格化，使得相同的代码出现的位置保持一致，易于比较变异体与待测程序之间的区别.

(2) 程序分块. 采用程序分析技术对待测程序进行块结构分析，得到 10 个程序块. 其中，b_2、b_4、b_5、b_8、b_9、b_{10} 是顺序块，b_1、b_3 是循环块，b_6、b_7 是选择块.

(3) 变异体分组. 获取变异体的变异位置，分析该位置所属的程序块，按程序块将变异体分为不同的组. 示例中，变异体 *mutant_0* 和 *mutant_1* 属于同一组.

(4) 程序合并. 对于每组中的变异体，使用程序合成方法将其与原程序合成为一个程序，使用子进程去执行变异体的变异块，加入对程序状态的输出，并及时结束执行. 图 10 示意了程序合并的结果，*__fpid* 区分原程序和变异体，*__count* 区分不同的变异块. 父进程 fork 每一子进程后等待子进程结束，即每个变异块依次执行.

(5) 计算中间状态. 使用测试用例去执行块程序，输出原程序和每个变异体的程序中间状态并进行比较，获取每个变异体的存活状态. 图 10 中的 saveState 是状态保存代码的简化表示，具体实现采用 C 库函数 fwrite 一一读取并保存状态可能改变的变量的内存数据，结合 sizeof 运算符在编译时获取变量的数据大小.

(6) 统计测试结果. 统计程序分块结果、变异体分组结果、变异体合成结果、测试执行结果等.

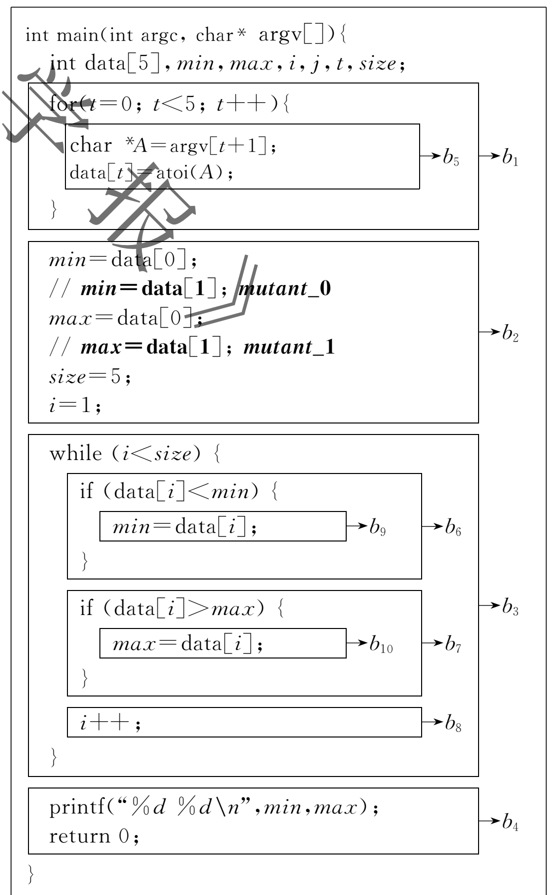

图 9　代码和变异体示例

计算机学报

```
int __fpid=-1;
int __count=-1;
int main(int argc, char *argv[]){
  int data[5], min, max, i, j, t, size;
  for(t=0; t<5; t++){
       char *A=argv[t+1];
       data[t]=atoi(A);
  }
  while(__fpid !=0 && ++__count<2){
      __fpid=fork();
      if(__fpid==0) break;
      wait(NULL);
  }
  if(__fpid==0){
      switch(__count){
      case 0:
          if(!isKilled(mutant_0){
              min=data[1];
              max=data[0];
              size=5;
              i=1;
              saveState(&min, &max, &size, &i);
          } else
              exit(0);
          break;
      case 1:
          if(!isKilled(mutant_1)){
              min=data[0];
              max=data[1];
              size=5;
              i=1;
              saveState(&min, &max, &size, &i);
          } else
              exit(0);
          break;
      default:
          exit(0);
      }
  } else {
      min=data[0];
      max=data[0];
      size=5;
      i=1;
      saveState(&min, &max, &size, &i);
  }
  exit(0);
  return 0;
}
```

图 10　程序合并示例

4　实验评估

弱变异测试[12]与并发变异测试[15]采用不同的方法加速变异执行速度，有效缩短传统变异测试的时间. 本文 3.2 节采用理论分析的方式，比较了两种基准技术（并发变异测试与弱变异测试）以及并发弱变异测试对传统变异测试的测试时间优化效率. 本章采用经验研究的方式，进一步评估所提方法的有效性与性能影响因素，比较本文方法与两种基准技术的优化效果.

4.1　研究问题

本文通过实验评估回答如下四个研究问题：

（1）并发弱变异测试能否减少变异测试的执行时间？

通过比较并发弱变异测试与传统变异测试的编译时间与执行时间，评价并发弱变异的优化效果.

（2）与并发变异测试和弱变异测试相比，并发弱变异测试的加速变异测试的效果如何？

通过实验评估三种变异测试优化技术的优化效率.

（3）并发弱变异测试对变异测试的有效性的影响？

并发弱变异测试基于弱变异测试加速变异测试执行，同时是否降低变异测试充分性评估效果. 通过实验评估并发弱变异测试对传统变异测试有效性的影响程度.

（4）影响并发弱变异测试技术优化效率的主要因素是什么？

通过实验验证与评估并发弱变异测试的优化效率的影响因素.

4.2　实验对象

本文采用三组 C 程序集作为实验对象：（1）科学计算程序集. minmax 获取一组数据的最大值和最小值；bubble 实现冒泡排序算法；triangle 是一个判断三角形类型的程序；nextdate 计算下一天的日期；（2）西门子程序集. tcas 是一个飞行器防碰撞程序；schedule 和 schedule2 是两个调度程序；tot_info 针对指定的输入数据生成统计信息；print_tokens 和 print_tokens2 是两个词法分析器；replace 完成模式匹配和替换；（3）space 程序集. 针对数组定义语言（ADL）的解释器. 表 1 总结了实验对象的基本信息.

表 1　实验程序的基本信息

实验对象	代码行数	测试用例数量	非等价变异体数量
minmax	33	60	115
bubble	24	75	149
triangle	28	188	175
nextdate	83	377	405
tcas	133	1608	1291
tot_info	245	938	1882
schedule	288	2650	740
schedule2	264	2710	1028
print_tokens	431	4071	1377
print_tokens2	380	4055	1945
replace	494	5542	4031
space	5981	13497	15467

本文实验评估所用测试用例集的构造分为两步：首先根据等价类划分、边界值分析设计部分测试用例；然后采用语句覆盖、分支覆盖、定义-使用覆盖

《计算机学报》

等补充测试用例. 针对每一个覆盖标准至少设计30个测试用例. 表1中"测试用例数量"一栏列出了每个实验程序的测试用例集的规模. 上述实验设置与并发变异测试[15]的评估方法一致,从而保证了与基准技术比较的公平性.

本文实验评估所用的变异体由Proteum[20]生成. Proteum支持操作符变异、变量变异、语句变异等常见的变异算子类型. 实验中,首先剔除无法通过编译的变异体,然后识别等价变异体. 等价变异体识别分为两步:首先对每个程序进行一次强变异测试,对于存活的变异体,采用人工分析的方式进一步判定其是否是等价变异体. 最后,将识别出的等价变异体从变异体集合中删除,得到非等价变异体集合.

实验运行环境的配置为Intel Core i7-10700 1.80GHz CPU、24GB内存、Ubuntu 20.04.4系统. 基于Clang编译框架实现C程序分析与合成,使用GCC编译C程序与变异体.

4.3 度量指标

本文采用编译加速比 CTA 和执行加速比 ETA 两个指标衡量测试优化技术的优化效率,采用变异得分评估并发弱变异测试对变异测试测试充分性的影响.

编译加速比 CTA 评估相应的优化技术缩短传统变异测试的编译时间的百分比,其计算公式如下:

$$CTA=\frac{TMCT-NMCT}{TMCT}\times 100\% \tag{2}$$

其中 $TMCT$ 表示传统变异测试的编译时间,$NMCT$ 表示采用弱变异、并发变异、并发弱变异等优化技术后变异体程序的编译时间.

执行加速比 ETA 表示相应的优化技术缩短传统变异测试的执行时间的百分比,其计算公式如下:

$$ETA=\frac{TMET-NMET}{TMET}\times 100\% \tag{3}$$

其中 $TMET$ 表示传统变异测试的执行时间,$NMET$ 表示采用变异优化技术后的执行时间.

相对变异得分 RMS 衡量并发弱变异测试相对于传统变异测试的故障检测能力,评估变异执行加速对故障检测能力的影响. 给定一个程序 p 与变异体集合 M,采用并发弱变异测试构造能够杀死弱变异体的测试用例子集 TS_{CWMT}. 对变异体集合 M 执行传统变异测试,执行测试用例集 TS_{CWMT} 并计算变异得分,称为"相对变异得分". 相对变异得分的计算公式如下:

$$RMS_p^M=\frac{|M_k|}{|M|}\times 100\% \tag{4}$$

其中,$|M_k|$ 为测试用例集 TS_{CWMT} 采用传统的变异测试时所杀死 M 中的变异体数量,$|M|$ 为总的变异体数量. 相对变异得分越高,表明并发弱变异的测试充分性评估有效性越高,变异执行加速对测试充分性评估的影响越小.

4.4 实验步骤

针对研究问题1和2,按以下步骤进行实验评估:

(1) 选取实验对象程序 p,选取该程序的所有非等价变异体(执行变异体,当变异体的测试用例执行结果与原程序执行结果不一致时为非等价变异体)作为变异体集合 M,选取所有的测试用例作为测试用例集 T;

(2) 采用每种变异测试技术(传统变异测试、并发变异测试、弱变异测试和并发弱变异测试)对变异体程序进行相应的处理,编译生成可执行程序,记录相应的编译时间;

(3) 重复(2)十次,计算每种变异测试技术的编译时间平均值;

(4) 随机打乱测试用例的顺序;

(5) 对每种变异测试技术,依次选取测试用例集 T 中的每个测试用例运行可执行程序,直至所有的变异体被杀死或执行完所有的测试用例,记录相应的执行时间;

(6) 重复(4)和(5)三十次,计算每种变异测试技术的执行时间平均值;

(7) 计算并发变异测试、弱变异测试和并发弱变异测试技术的编译加速比 CTA 和执行加速比 ETA;

(8) 重复(1)到(7),直至每个实验对象程序都被测试一遍.

针对研究问题3,按以下步骤进行实验评估:

(1) 选取实验对象程序 p,选取该程序的所有非等价变异体作为变异体集合 M,选取所有的测试用例作为测试用例集 T;

(2) 生成一个空的测试用例子集 T';

(3) 使用并发弱变异测试方法,将变异体合成为块程序;

(4) 从测试用例集中的随机选取一个测试用例运行块程序;

(5) 若该测试用例能够杀死块程序中未被杀死过的变异体,则将该测试用例加入测试用例子集 T';

(6) 重复(4)和(5),直至所有块程序中的变异体被杀死或执行完所有的测试用例;

(7) 将测试用例子集 T' 用于传统变异测试;

《计算机学报》

(8) 统计变异体的存活情况，计算相对变异得分；

(9) 重复(2)到(8)十次，计算相对变异得分的平均值.

针对研究问题4，按以下步骤进行实验评估：

统计程序 p 中程序块的数量和变异体集合的数量，分析程序块的平均变异体数量与并发弱变异测试优化效率的相关性.

4.5　实验结果与分析

(1) 并发弱变异测试的有效性分析. 表2和表3分别总结了传统变异测试、并发变异测试、弱变异测试与并发弱变异测试四种测试技术应用于12个实验程序的编译时间与执行时间，以及后三种测试技术相对于传统变异测试技术的加速比. 其中，并发变异测试、弱变异测试和并发弱变异测试的平均编译加速比分别为92.38%、−9.30%和92.22%，平均执行加速比分别为50.53%、48.79%和71.22%. 实验结果表明：弱变异测试会增加编译时间，但是并发变异测试和并发弱变异测试的编译时间均明显小于传统变异测试，且并发弱变异测试在执行时间加速上优于并发变异测试、弱变异测试两种基线技术，即并发弱变异测试可以有效提升传统变异测试的速度.

表2　编译时间与编译加速比

实验对象	传统变异测试时间/s	并发变异测试		弱变异测试		并发弱变异测试	
		时间/s	加速比/%	时间/s	加速比/%	时间/s	加速比/%
minmax	3.261	0.337	89.67	3.675	−12.70	0.344	89.45
bubble	4.221	0.287	93.20	4.612	−9.26	0.292	93.08
triangle	4.921	0.276	94.39	5.305	−7.80	0.287	94.17
nextdate	11.838	0.924	92.19	13.053	−10.26	0.952	91.96
schedule	27.593	3.173	88.50	31.162	−12.93	3.234	88.28
schedule2	39.272	3.357	91.45	43.230	−10.08	3.424	91.28
tcas	41.213	1.436	96.52	43.322	−5.12	1.460	96.46
print_tokens	56.011	4.890	91.27	61.888	−10.49	4.994	91.08
print_tokens2	74.500	7.119	90.44	82.517	−10.76	7.247	90.27
tot_info	84.569	4.007	95.26	89.550	−5.89	4.078	95.18
replace	174.247	9.388	94.61	186.439	−7.00	9.548	94.52
space	3161.914	283.198	91.04	3457.048	−9.33	287.993	90.89

表3　执行时间与执行加速比

实验对象	传统变异测试时间/s	并发变异测试		弱变异测试		并发弱变异测试	
		时间/s	加速比/%	时间/s	加速比/%	时间/s	加速比/%
minmax	0.515	0.199	61.36	0.283	45.05	0.175	66.02
bubble	0.703	0.593	15.65	0.188	73.26	0.174	75.25
triangle	1.961	0.874	55.43	1.857	5.30	0.849	56.71
nextdate	2.060	0.809	60.73	1.050	49.03	0.426	79.32
schedule	45.985	31.932	30.56	32.980	28.28	21.596	53.04
schedule2	183.621	91.951	49.92	64.001	65.15	45.039	75.47
tcas	28.635	10.719	62.57	12.896	54.96	5.075	82.28
print_tokens	108.106	52.779	51.18	32.517	69.92	22.079	79.58
print_tokens2	131.014	58.182	55.59	105.562	19.43	47.513	63.73
tot_info	39.256	19.996	49.06	20.044	48.94	12.877	67.20
replace	210.941	147.908	29.88	111.167	47.30	72.320	65.72
space	7419.321	1152.435	84.47	1571.678	78.82	713.032	90.39

(2) 并发弱变异测试的优化效率分析. 编译时间与待编译程序的数量和程序逻辑相关. 并发变异测试、弱变异测试、并发弱变异测试三种变异测试优化技术的编译加速比如图11所示. 在弱变异测试中，为了获取程序的中间状态，对程序代码进行了额外处理，其编译时间比传统变异测试的编译时间长，因此编译加速比为负值. “CMT”表示并发变异测试；“WMT”表示弱变异测试；“CWMT”表示并发弱变异测试. 实验结果表明：① 并发变异测试对变异测试的编译加速具有显著作用，主要原因是并发程序合成减少了大量的冗余代码的编译时间；② 弱变异测试增加了编译时间，主要原因是状态保存代码引入了额外的编译开销；③ 并发弱变异测试在显著提升编译效率的同时，有效缓解了弱变异测试的编译时间开销.

测试执行时间与测试执行的次数和单次测试执行的时间相关. 并发变异测试可以减少测试执行的次数，但是没有减少单次测试执行的时间，因为在一

《计算机学报》

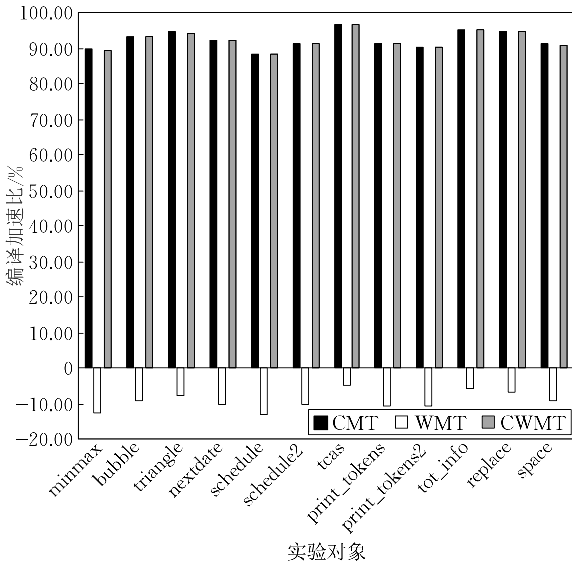

图 11 编译加速比

次执行时需要执行变异体的所有程序块；弱变异测试可以减少单次测试执行的时间；并发弱变异测试可以同时减少测试执行的次数和单次执行的时间. 并发变异测试、弱变异测试、并发弱变异测试三种变异测试优化技术的执行加速比如图 12 所示. "CMT"表示并发变异测试；"WMT"表示弱变异测试；"CWMT"表示并发弱变异测试. 其中，并发变异测试和弱变异测试分别在 4 个实验对象上达到 60%以上的执行加速比；并发弱变异测试在 6 个实验对象上达到 70%以上的执行加速比. 三种变异测试技术在大规模程序 space 上的执行优化效果最好，CMT 在 bubble、WMT 在 triangle、CWMT 在 schedule 上的执行优化效果最差. 并发变异测试和弱变异测试在不同实验对象上表现出不一致的执行加速比；并发弱变异测试在所有实验对象上均表现出最高的执行加速比，且加速比都超过 50%.

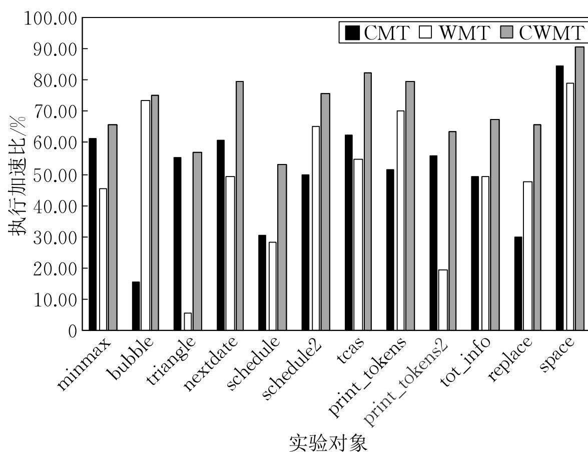

图 12 执行加速比

实验结果表明：① 三种变异测试优化技术中，并发弱变异测试在变异执行加速上有显著的优越性，而并发变异测试和弱变异测试基本相当；② 随着程序规模的增加，并发弱变异测试的执行加速比有所提升，即并发弱变异测试有助于大规模程序的变异测试的执行加速；③ 并发变异测试和弱变异测试的优化效果越好，则并发弱变异测试的执行加速比越高.

并发弱变异测试的执行加速比除了与变异测试技术相关，也与实验对象的程序结构有关. 具体说来，triangle 主要由条件判断和输出语句组成，呈现树型结构，程序入口到终点的距离较短，因此弱变异测试的优化效果有限，并发弱变异测试的执行加速主要由并发机制获得；bubble 程序虽然规模较小，但由多个计算密集的循环语句构成，程序入口到终点的距离较长，弱变异测试的优化机制对并发弱变异测试的执行加速比提升更加突出.

(3) 并发弱变异测试对变异测试有效性的影响分析. 与弱变异测试一样，并发弱变异测试对变异体杀死进行了弱化处理，采用程序中间状态差异代替程序的输出差异对变异体是否杀死进行判断，不考虑错误状态的传播. 由于程序中的某些变量的改变不一定影响程序的最终输出，能够杀死弱变异体的测试用例不一定能杀死相应的强变异体. 并发弱变异测试的充分性评估的有效性可能受影响.

表 4 总结了并发弱变异测试的相对变异得分评估结果，衡量并发弱变异测试相对于传统变异测试的故障检测能力，其中 TS_{ALL} 表示测试用例全集，TS_{CWMT} 表示杀死并发弱变异体的测试用例子集，TS_{CWMT} 表示杀死的变异体数和相应的相对变异得分取十次测试的平均值. 具体说来：① 12 个实验对象中，11 个实验对象的变异得分有所下降，平均降

表 4 相对变异得分

实验对象	TS_{ALL} 杀死变异体数	变异得分/%	TS_{CWMT} 杀死变异体数	相对变异得分/%
minmax	115	100.00	111.8	97.22
bubble	149	100.00	148.7	99.80
triangle	175	100.00	175.0	100.00
nextdate	405	100.00	385.7	95.23
schedule	740	100.00	733.7	99.15
schedule2	1027	99.90	934.2	90.88
tcas	1291	100.00	1235.9	95.73
print_tokens	1377	100.00	1367.9	99.34
print_tokens2	1945	100.00	1943.1	99.90
tot_info	1882	100.00	1839.3	97.73
replace	3889	96.48	3745.1	92.91
space	13835	89.45	13560.7	87.68

低了 2.52%，表明并发变异测试在加速变异执行的同时，对测试充分性评估有一定的影响；② 应用于不同的实验对象时，并发弱变异测试的相对变异得分的差别较大. 例如，schedule2 和 space 的变异得分分别降低了 9.03% 和 1.77%，表明并发弱变异测试对测试充分性评估的影响在不同程序上表现不同.

需要指出的是，弱变异测试面临的测试充分性评估的有效性减弱问题，由于并发弱变异测试继承了弱变异测试的基本原理，并发弱变异测试也同样存在，体现为相对变异得分的降低. 另一方面，并发弱变异测试更加侧重对弱变异测试的执行加速.

(4) 并发弱变异测试优化效率影响因素分析. 并发弱变异测试将变异位置属于相同程序块的所有变异体合成为一个块程序. 不难想象，属于同一个程序块中的变异体的数量越多(即块变异体密度大)，程序合成得到的程序数则减少得越多. 因此，我们猜想被测程序的块变异体密度(即每个块中变异体数量的平均值)是影响并发弱变异测试的变异体编译和变异执行的优化效率之一. 表 5 总结了所有实验对象的程序块信息以及加速比信息.

表 5　实验对象程序块信息

实验对象	块总数	基本块		选择块		循环块		块变异体密度	并发弱变异测试	
		数量	占比/%	数量	占比/%	数量	占比/%		编译加速比/%	执行加速比/%
minmax	18	14	77.78	2	11.11	2	11.11	6.39	89.45	66.02
bubble	13	8	61.54	1	7.69	4	30.77	11.46	93.08	75.25
triangle	19	16	84.21	3	15.79	0	0.00	9.21	94.17	56.71
nextdate	50	45	90.00	5	10.00	0	0.00	8.10	91.96	79.32
schedule	151	126	83.44	19	12.58	6	3.97	4.90	88.28	53.04
schedule2	163	132	80.98	23	14.11	8	4.91	6.31	91.28	75.47
tcas	54	49	90.74	5	9.26	0	0.00	23.91	96.46	82.28
print_tokens	285	253	88.77	25	8.77	7	2.46	4.83	91.08	79.58
print_tokens2	228	160	70.18	61	26.75	7	3.07	8.53	90.27	63.73
tot_info	128	95	74.22	19	14.84	14	10.94	14.70	95.18	67.20
replace	257	213	82.88	31	12.06	13	5.06	15.68	94.52	65.72
space	1697	1167	68.77	478	28.17	52	3.06	9.11	90.89	90.39

首先分析块变异体数量的平均值对变异优化效率的影响. 图 13 展示了块变异体密度和编译加速比的关系. 结果表明，绝大多数情况下，程序的块变异体密度越大时，并发弱变异测试的编译加速效果越明显. 图 14 展示了块变异体密度与执行加速比的关系. 由于并发弱变异引入了并发处理机制，并发处理的相关代码与原始程序、变异体的原始代码的执行时间一起构成了合成后的程序执行时间. 不同程序的处理逻辑的复杂性不同，并发处理的相关代码引入的执行时间占比不确定，块变异体密度对执行加速比的影响规律不明显.

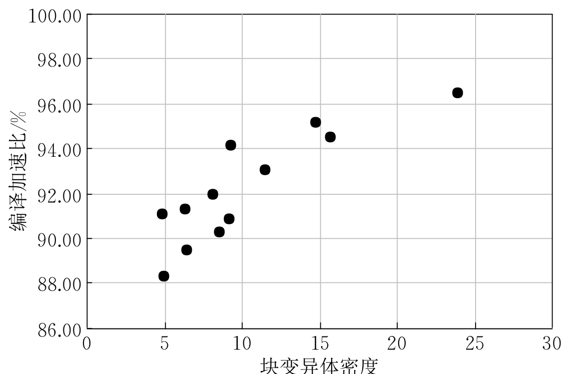

图 13　编译加速比与块变异体密度关系

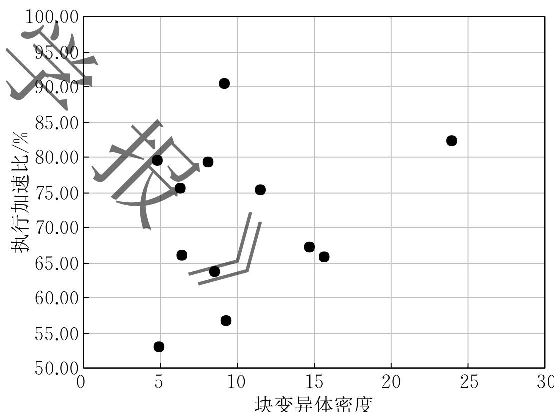

图 14　执行加速比与块变异体密度关系

5　相关工作

变异测试是一种具有较强故障检测能力的测试技术，但是高昂的测试代价限制了其在测试实践中的广泛应用. 为了降低变异测试的代价，使其可应用于大型程序，人们从减少变异体数量和缩短变异体执行时间两个角度，提出了多种变异测试优化技术[3].

从减少变异体数量角度缩短测试时间时，核心问题是如何选择具有代表性的变异体子集. 人们提

出了多种变异体选择策略.

随机选择策略从变异体集合中随机选取子集,实验结果表明,随机选择 10%变异体时,变异测试的有效性降低不超过 30%;若选择 50%变异体,其有效性降低不超过 10%[21].

变异算子选择策略通过有选择地使用变异算子产生不同类别的变异体,生成较小的变异体子集. Siami 等人[22]使用一个较小的变异算子集合评估了该策略产生的变异体集合的变异得分变化情况,实验结果表明在变异体集缩减 93%的情形下并不显著影响变异得分. Álvarez-García 等人[23]基于冗余度和测试强度两个指标选取 C++程序的变异算子,在减少变异体数量的同时保证变异测试的有效性.

基于程序结构的变异体选择策略依据不同的程序结构性指标选择部分变异体进行变异测试. Sun 等人[16]提出了一种基于控制流的变异体约简技术,优先选择变异位置所在的程序路径更深的变异体,能够更早发现测试用例的不足. Mirshokraie 等人[24]通过指定变异算子的作用范围来减少变异体数量,利用静态和动态分析技术计算程序中函数和分支的重要性,选择更重要的函数或分支进行变异操作. Patrick 等人[25]基于符号执行选取和原程序语义相近的变异体,实验结果表明这类变异体难以被一般的测试用例杀死. Anthony 等人[26]提出一种增量变异测试方法,将变异算子的作用范围限制在变更的代码片段,从而大幅度减少变异体的数量.

冗余变异体识别与删除策略通过分析不同变异体之间存在的检测蕴含关系,识别冗余变异体并将其从变异体集合中删除,从而减少变异体的数量. Sun 等人[27]依据程序中的变量定值-使用关系定义了一组冗余变异体识别规则,采用静态程序分析技术识别并删除冗余变异体,减少变异测试的执行时间. 为了进一步提高冗余变异体识别效果,Sun 等人[28]提出了一种基于符号执行的冗余变异体识别方法,使用选择符号执行技术分析不同变异体在执行变异点后的程序状态变化,依据程序状态的相似性识别冗余变异体,实验结果表明该方法相比静态分析方法识别出了更多的冗余变异体.

高阶变异体策略通过对待测程序进行多次变异产生高阶变异体,按阶数成比例地减少变异体数量. Polo 等人[29]通过三种生成策略对一阶变异体进行两两组合得到二阶变异体,使变异体数量缩减一半,能够在不降低测试用例强度情况下显著降低测试代价. Jia 等人[30]基于启发式搜索生成强包含高阶变异体,这种变异体比其所包含的任何一阶变异体都更难被杀死. Abuljadayel 等人[31]基于遗传算法生成高阶变异体,通过适应度函数进化出更难被杀死的高阶变异体,同时保证产生较少的等价变异体.

上述变异体约简策略虽然通过减少变异体的数量有效降低了变异测试的代价,但选择的变异体子集会对测试充分性评估方面有所影响,例如导致变异得分准确性的下降[9]. 本文提出的并发弱变异测试技术并不减少变异体的数量,而是对变异体进行了合成. 与高阶变异体的合成机制不同的是,并发弱变异测试不改变原先变异体的执行逻辑,且可以单独判断每个变异体是否被杀死. 降低变异测试开销的第二类方案是缩短变异体的编译时间和执行时间.

基于编译的变异测试加速方法通过采用编译优化技术减少变异体的编译时间,降低变异测试开销. Ma 等人[32]基于字节码翻译技术开发了 Java 变异测试工具 MuJava,修改 Java 原程序的字节码得到变异体,避免了变异体源代码到字节码的编译过程. Android 变异测试工具 MutAPK[33]对 APK 进行解包获取 Android 程序的 Smali 字节码,对字节码变异后再重新打包,避免了编译过程. 面向多种编程语言的变异测试工具 Mull[34]对原程序通过 LLVM 编译框架生成的中间代码进行变异,只有变异代码片段需要重新编译. Mart[35]直接对 LLVM 中间代码的二进制形式 Bitcode 进行变异,同时也通过静态程序分析技术来避免一些等价变异体的生成,进一步减少了编译时间. 并发弱变异测试中的程序合并步骤减少了大量重复代码的编译,显著提高了编译效率.

基于测试用例的变异测试加速方法对测试用例进行排序或选取[36],从而节省变异测试的时间. Just 等人[37]根据运行时差异对测试用例进行排序,优先选择在原程序上执行更快的测试用例. Zhang 等人[38]通过优先选择可能杀死变异体的测试用例来加快变异测试的速度,首先在变异测试前根据测试用例在原程序上的执行情况对覆盖较多次变异位置的或覆盖的变异位置接近程序出口的测试用例赋予高优先级,然后在测试过程中优先选择杀死较多与待执行变异体相似的变异体的测试用例. 变异体被杀死的必要条件是测试用例触发变异点,基于该思想,Papadakis 和 Malevris[39-40]首先记录每个测试用

例在原程序上运行的语句块覆盖信息，然后在变异测试过程中，仅选择能够覆盖变异语句块的测试用例．另一方面，测试用例顺序是变异测试执行时间的重要影响因素．本文对比实验中按照随机的测试用例顺序进行多次重复实验，可以缓解该因素对评估结果的影响．

预测性变异测试基于大量的变异执行信息训练分类模型预测变异体能否被杀死，在不执行变异体的情况下估计变异得分．Zhang 等人[14]基于变异体的程序属性和历史执行信息提取特征，构建并训练随机森林分类器，该方法可以预测超过 60％的变异体能否被杀死，从而缩短变异测试的时间．

弱变异测试通过减少变异点后的执行过程加速变异测试．Howden 首次提出弱变异测试技术[12]，通过分析变异点后的程序状态来判断变异体是否被杀死，减少了变异点后代码片段的执行．Offutt 与 Le[2]根据判断程序状态的时机和位置的不同扩展了弱变异测试技术，并引入等价变异体识别进一步提高弱变异测试效率．Just 等人[41]提出测试等价的概念，基于对程序状态传播的分析来优化变异测试执行，可以缩短 40％的变异测试执行时间．对于某个测试用例的执行，变异体和原程序是测试等价的，则二者变异点后的程序状态没有差异或差异只是局部存在，该变异体不需要继续执行；变异体之间是测试等价的，则它们在变异点后的程序状态相同，只有其中之一需要继续执行．

待测程序和变异体在变异点前存在共同执行的部分，可以减少这部分不必要的执行来加速变异测试．基于该思想，Sun 等人[15]提出了并发变异测试技术，采用程序分析与合成技术将位于某个程序块的多个变异体合成，合成的变异体程序减少了变异点之前的重复执行，通过并发机制继续之后的执行，从而加速变异测试的执行过程．Wang 等人[42]同时优化变异点前后的不必要执行，把不同变异体中变异点前相同执行过程的部分进行共享，然后去执行变异点后不同的部分，将变异点后程序状态相同的多个变异体归为同一个等价类，选择其中之一继续执行．不同于上述方法在变异体执行时进行程序状态监控和动态处理，并发弱变异测试在变异体执行前进行相关的代码优化．

人们还尝试通过并行计算提高变异测试的效率．Mathur 和 Krauser[43]将变异测试与并行优化相结合，将标量替换变异体的执行转换为向量处理器上的并行计算，验证了并行变异测试的可行性．Mateo 等人[44]提出了一种并行变异测试技术，在分布式系统上并行执行变异体，采用五种工作负载分配算法对变异体的执行过程进行动态的并行优化．并行变异测试技术虽然能够显著提高变异测试效率，但依赖于特定的运行环境．与上述技术相比，并发弱变异测试对运行环境没有特别要求．

本文提出的并发弱变异测试技术采用程序合成技术与并发机制将同属某个程序块的一组变异体合成，减少变异点之前代码块的重复执行，通过比较程序中间状态来判断变异体是否被杀死，缩减变异点到程序结束之间代码块的执行，进一步加速变异执行．本文方法是并发变异测试与弱变异的有效融合，不仅提升了变异测试的执行加速，而且解决了并发机制应用于弱变异测试的关键问题．

6　结　论

变异测试广泛用于评价测试用例集充分性和软件测试技术有效性．由于计算开销大等原因，变异测试难以在软件测试实践中广泛使用．本文从加速变异测试执行的角度出发，探索变异测试优化技术．在弱变异测试的基础上，提出了一种并发弱变异测试技术，结合程序合成与并发机制减少一组变异体之间存在的相同代码片段的重复执行以及变异点之后代码片段的执行，不仅缩短了变异体的编译时间，也缩短了变异测试的执行时间．采用了多个实验程序评估了本文方法的有效性，实验结果表明：(1) 并发弱变异测试有效地减少了传统变异测试的编译时间与执行时间；(2) 并发弱变异测试在提升变异测试效率的同时，也一定程度影响了变异测试的充分性评估能力；(3) 影响并发弱变异测试优化效率的主要因素之一是块变异体密度．

并发弱变异测试提供了一种新的变异测试优化方法．未来的工作包括：(1) 本文重点解决了并发弱变异测试应用于 C 程序时的关键问题，由于不同类型程序适用的并发机制与程序合成技术不同，探索将本文方法推广到其它类型的程序（如 Java 程序）；(2) 将并发弱变异测试与测试用例选择策略、预测性变异测试等结合，探索更加高效的变异测试优化技术，进一步降低变异测试的执行开销．

参　考　文　献

[1]　Papadakis M, Kintis M, Zhang J, et al. Mutation testing

《计算机学报》

advances: An analysis and survey. Advances in Computers. Elsevier, 2019, 112(1): 275-378

[2] Offutt A J, Lee S D. An empirical evaluation of weak mutation. IEEE Transactions on Software Engineering, 1994, 20(5): 337-344

[3] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering, 2010, 37(5): 649-678

[4] Offutt J, Untch R H. Mutation 2000: Uniting the orthogonal //Proceedings of the Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries. California, USA, 2000: 45-55

[5] Ferrari F C, Pizzoleto A V, Offutt J. A systematic review of cost reduction techniques for mutation testing: Preliminary results//Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). Västerås, Sweden, 2018: 1-10

[6] Bluemke I, Kulesza K. Reduction of computational cost in mutation testing by sampling mutants//Proceedings of the 8th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. Brunow, Poland, 2013: 41-51

[7] Bluemke I, Kulesza K. Reduction in mutation testing of Java classes//Proceedings of the 9th International Conference on Software Engineering and Application (ICSOFT-EA). Vienna, Austria, 2014: 297-304

[8] Harman M, Jia Y, Mateo P R, et al. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation//Proceedings of the 29th IEEE International Conference on Automated Software Engineering (ASE 2014). Vasteras, Sweden, 2014: 297-408

[9] Gopinath R, Ahmed I, Alipour M A, et al. Mutation reduction strategies considered harmful. IEEE Transactions on Reliability, 2017, 66(3): 854-874

[10] Tuya J, Suárez Cabal M J, Riva Álvarez C. Mutating database queries. Information and Software Technology, 2007, 49(4): 398-417

[11] Delgado-Pérez P, Medina-Bulo I, Merayo M G. Using evolutionary computation to improve mutation testing//Proceedings of the 14th International Work-Conference on Artificial Neural Networks (IWANN). Springer, Cham, 2017: 381-391

[12] Howden W E. Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering, 1982, 8(4): 371-379

[13] Li N, West M, Escalona A, Durelli S. Mutation testing in practice using ruby//Proceedings of the 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW). Graz, Austria, 2015: 1-6

[14] Zhang J, Wang Z, Zhang L, et al. Predictive mutation testing. IEEE Transactions on Software Engineering, 2019, 45(9): 898-918

[15] Sun C, Jia J, Liu H, et al. A lightweight program dependence based approach to concurrent mutation analysis//Proceedings of the 42nd Annual Conference on Computer Software and Applications (COMPSAC 2018). Tokyo, Japan, 2018: 116-125

[16] Sun C, Xue F, Liu H, Zhang X. A path-aware approach to mutant reduction in mutation testing. Information & Software Technology, 2017, 81(C): 65-81

[17] Sun C, Jin M, Liu C, Jin R. An approach to static prediction and visual analysis of program execution time. Journal of Software, 2003, 14(1): 68-75 (in Chinese)
(孙昌爱, 金茂忠, 刘超, 靳若明. 程序执行时间的静态预估与可视化分析方法. 软件学报, 2003, 14(1): 68-75)

[18] Sun C, Liu C, Jin M. Effective wove algorithm for software structure graph. Journal of Beijing University of Aeronautics and Astronautics, 2000, 26(2): 705-709 (in Chinese)
(孙昌爱, 刘超, 金茂忠. 一种有效的软件结构图的布图算法. 北京航空航天大学学报, 2000, 26(2): 705-709)

[19] Yu M, Ma Y S. Possibility of cost reduction by mutant clustering according to the clustering scope. Software Testing, Verification and Reliability, 2019, 29(1-2): 1-24

[20] Delamaro M E, Maldonado J C, Vincenzi A M R. Proteum/IM 2.0: An integrated mutation testing environment//Wong W E ed. Mutation Testing for the New Century. Boston, MA: Springer, 2001: 91-101

[21] Papadakis M, Malevris N. An empirical evaluation of the first and second order mutation testing strategies//Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICST 2010). Paris, France, 2010: 90-99

[22] Siami N A, Andrews J H, Murdoch D J. Sufficient mutation operators for measuring test effectiveness//Proceedings of the 30th International Conference on Software Engineering (ICSE 2008). New York, USA, 2008: 351-360

[23] Álvarez-García M Á. Automation and evaluation of mutation testing for the new C++ Standards//Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). Madrid, Espana, 2021: 150-152

[24] Mirshokraie S, Mesbah A, Pattabiraman K. Guided mutation testing for JavaScript Web applications. IEEE Transactions on Software Engineering, 2015, 41(5): 429-444

[25] Patrick M, Oriol M, Clark J A. MESSI: Mutant evaluation by static semantic interpretation//Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST 2012). Montreal, Quebec Canada, 2012: 711-719

[26] Anthony Cachia M, Micallef M, Colombo C. Towards incremental mutation testing. Electronic Notes in Theoretical Computer Science, 2013, 294(1): 2-11

计算机学报

[27] Sun C, Guo X, Zhang X, Chen T Y. A data flow analysis based redundant mutants identification technique. Chinese Journal of Computers, 2019, 42(1): 44-60(in Chinese)
(孙昌爱，郭新玲，张翔宇，陈宗岳. 一种基于数据流分析的冗余变异体识别方法. 计算机学报，2019，42(1)：44-60)

[28] Sun C, Fu A, Guo X, Chen T Y. ReMuSSE: A redundant mutant identification technique based on selective symbolic execution. IEEE Transactions on Reliability, 2022, 71(1): 415-428

[29] Polo M, Piattini M, García Rodríguez I. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification and Reliability, 2009, 19(2): 111-131

[30] Jia Y, Harman M. Higher order mutation testing. Information and Software Technology, 2009, 51(10): 1379-1393

[31] Abuljadayel A, Wedyan F. An approach for the generation of higher order mutants using genetic algorithms. International Journal of Intelligent Systems and Applications, 2018, 11(1): 34-45

[32] Ma Y S, Offutt J, Kwon Y R. MuJava: An automated class mutation system. Software Testing, Verification and Reliability, 2005, 15(2): 97-133

[33] Escobar-Velásquez C, Osorio-Riaño M, Linares-Vásquez M. MutAPK: Source-codeless mutant generation for android apps//Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering(ASE 2019). California, USA, 2019: 1090-1093

[34] Denisov A, Pankevich S. Mull it over: Mutation testing based on LLVM//Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation Workshops(ICSTW). Västerås, Sweden, 2018: 25-31

[35] Chekam T T, Papadakis M, Le Traon Y. Mart: A mutant generation tool for LLVM//Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE 2019). New York, USA, 2019: 1080-1084

[36] Shen W, Li Y, Han Y, et al. Boundary sampling to boost mutation testing for deep learning models. Information and Software Technology, 2020, 130(1): 1-16

[37] Just R, Kapfhammer G M, Schweiggert F. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis//Proceedings of the 23rd International Symposium on Software Reliability Engineering(ISSRE 2012). Dallas, USA, 2012: 11-20

[38] Zhang L, Marinov D, Khurshid S. Faster mutation testing inspired by test prioritization and reduction//Proceedings of the 7th International Symposium on Software Testing and Analysis(ISSTA 2013). Roma, Italy, 2013: 235-245

[39] Papadakis M, Malevris N, Kallia M. Towards automating the generation of mutation tests//Proceedings of the 5th Workshop on Automation of Software Test (AST). Cape Town, South Africa, 2010: 111-118

[40] Papadakis M, Malevris N. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Software Quality Journal, 2011, 19(4): 691-723

[41] Just R, Ernst M D, Fraser G. Efficient mutation analysis by propagating and partitioning infected execution states//Proceedings of the 8th International Symposium on Software Testing and Analysis (ISSTA 2014). California, USA, 2014: 315-326

[42] Wang B, Xiong Y, Shi Y, et al. Faster mutation analysis via equivalence modulo states//Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA 2017). California, USA, 2017: 295-306

[43] Mathur A P, Krauser E W. Modeling mutation on a vector processor//Proceedings of the 10th International Conference on Software Engineering (ICSE 1988). Singapore, 1988: 154-161

[44] Mateo P R, Usaola M P. Parallel mutation testing. Software Testing, Verification and Reliability, 2013, 23(4): 315-350

《计算机学报》

SUN Chang-Ai, Ph. D., professor, Ph. D. supervisor. His research interests include software testing, program analysis, software architecture, and service-oriented computing.

ZENG Guo-Feng, Ph. D. candidate. His research interest is software testing.

ZHANG Shou-Feng, M. S. candidate. His research interest is software testing.

TANG Jin, M. S. candidate. His research interest is software testing.

LI Ning, M. S., senior engineer. His research interest is software testing.

ZHANG Shi-Yong, bachelor, engineer. His research interest is software testing.

CHEN Yan, M. S., senior engineer. Her research interest is information systems.

Background

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 and a long testing period prevents mutation testing from being widely adopted in practice. We observe that the bulk of codes in multiple mutants are repeatedly executed through an analysis of the mutation testing process. In this paper, we explore the improvement of mutation testing in terms of reducing its execution time and accordingly propose a concurrent weak mutation testing approach, which provides an optimal mechanism to removal of redundant executions by using the combination of concurrent controls and program synthesis techniques, aiming at shortening the execution time of mutation testing. An empirical study has been conducted in which 12 C programs were used as subject program to evaluate the effectiveness and efficiency of the proposed approach and analyze the impact factors of performance optimization. Experimental results show that the proposed approach can significantly improve the efficiency of mutation testing, with an average reduction rate of more than 50% for all subject programs and up to 99% for those large-scale programs. The proposed technique provide a new solution to accelerate mutation testing without the need of special infrastructure. It is also very promising to explore the integration with existing optimized techniques to deliver an even faster mutation testing technique.

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61872039 and 62272037, the Aeronautical Science Foundation of China under Grant No. 2016ZD74004, the Fundamental Research Funds for the Central Universities under Grant No. FRF-GF-19-19B, and the joint project under Grant No. 19010203 with North China Institute of Computing Technology.

《计算机学报》