软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*, 2022,33(2):397−409 [doi: 10.13328/j.cnki.jos.006137]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# CMuJava: 一个面向 Java 程序并发变异体生成系统*

孙昌爱[1,2], 耿 宁[1], 代贺鹏[1], 顾友达[1]

[1](北京科技大学 计算机与通信工程学院, 北京 100083)

[2](计算机科学国家重点实验室(中国科学院 软件研究所), 北京 100190)

通信作者: 孙昌爱, E-mail: casun@ustb.edu.cn

**摘 要**: 并发程序由多个共享存储空间并发执行的流程组成. 由于流程之间执行次序的不确定性, 使得并发软件系统的测试比较困难. 变异测试是一种基于故障的软件测试技术, 广泛用于评估测试用例集的充分性和测试技术的有效性. 将变异测试应用于并发程序的一个关键问题是, 如何高效地生成大量的模拟并发故障的变异体集合. 给出了一种并发程序的变异测试框架, 研制了一个并发变异体自动生成工具 CMuJava. 采用经验研究的方式评估了 CMuJava 生成的变异体集合的正确性与充分性, 并且评估了变异体生成的效率. 实验结果表明: CMuJava 能够准确、充分地生成并发变异体集合, 极大地提高了手工变异体生成的效率.

**关键词**: 并发程序; 变异测试; 并发变异算子; 并发变异体; 测试工具

**中图法分类号**: TP311

中文引用格式: 孙昌爱, 耿宁, 代贺鹏, 顾友达. CMuJava: 一个面向 Java 程序并发变异体生成系统. 软件学报, 2022, 33(2): 397–409. http://www.jos.org.cn/1000-9825/6137.htm

英文引用格式: Sun CA, Geng N, Dai HP, Gu YD. CMuJava: Concurrent Mutant Generation System for Java. Ruan Jian Xue Bao/ Journal of Software, 2022, 33(2): 397–409 (in Chinese). http://www.jos.org.cn/1000-9825/6137.htm

## CMuJava: Concurrent Mutant Generation System for Java

SUN Chang-Ai[1,2], GENG Ning[1], DAI He-Peng[1], GU You-Da[1]

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

[2](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

**Abstract**: Concurrent programs are composed of multiple concurrent execution flows, which usually share storage space in an explicit or implicit manner. Uncertainty in the execution order of flows poses challenges for concurrent program testing. Mutation testing is a fault-based testing technique that is widely adopted to evaluate the adequacy of test suites and the effectiveness of test techniques. A key issue to applying mutation testing to concurrent programs is how to efficiently derive a large number of mutants that simulate possible concurrency-specific faults. This study proposes a mutation testing framework for concurrent programs and presents an automated concurrent mutant generation system called CMuJava. An empirical study is conducted to evaluate the correctness and adequacy of mutant sets generated by CMuJava and the mutant generation efficiency of CMuJava. The experimental results show that CMuJava can not only generate correct and adequate mutants, but also significantly improve the efficiency of manual mutant generation.

**Key words**: concurrent programs; mutation testing; concurrent mutation operators; concurrent mutants; testing tools

随着多核计算的日益普及, 并发程序引起了人们的广泛关注. 并发程序包含多个执行次序不确定的并发执行的流程, 这些流程之间显式或隐式地共享存储空间[1]. 这些并发流程的执行次序不确定情形称为执行交错. 执行交错使得完全相同的输入的输出结果可能不相同, 从而导致并发程序的故障难以重现. 如何有效地

* 基金项目: 国家自然科学基金(61872039); 中国科学院软件研究所计算机科学国家重点实验室开放课题(SYSKF1803); 中央高校基本科研业务费专项资金(FRF-GF-19-019B)

收稿时间: 2019-10-12; 修改时间: 2019-12-22; 采用时间: 2020-01-19

检测并发程序中潜藏的故障, 提高并发程序的可靠性, 是软件测试领域的一个难题.

变异测试[2]是一种基于故障的软件测试技术. 针对给定的待测程序, 通过应用变异算子来生成模拟故障的变异体, 然后采用测试用例检测这些变异体, 杀死的变异体的数量与所有非等价变异体数量的比值称为变异得分[3]. 变异得分广泛用来评估测试用例集的测试充分性与测试技术的有效性. 由于存在执行交错, 并发程序的充分性测试更加困难. 针对上述特点, 人们提出了一系列并发程序的测试技术. 例如, 锁集分析[17]、happens-before[18]和并发变异测试[4]. 其中, 并发变异测试将变异测试应用到并发程序, 通过变异算子模拟并发故障, 可以有效地提高并发程序的故障检测效率.

在实施并发变异测试时, 一个关键问题是如何生成大量模拟并发故障的并发变异体. 遗憾的是, 现有的并发变异测试支持工具存在各种各样的局限, 例如 Paraμ[9]支持 3 种并发变异算子、Comutation[10]不支持开源使用. 另一方面, 众多的面向 Java 程序的变异测试工具(例如 MuJava[5]、Javalance[6]、Jumble[7]和 Major[8]等)尚不支持并发变异体的生成.

MuJava 是一个具有代表性的面向 Java 程序的变异测试工具[11−13], 但不支持并发变异体. 本文通过扩展 MuJava 设计并实现了一个并发变异体生成工具 CMuJava, 以提高并发 Java 程序的变异测试的自动化程度和效率. 具体地,

(1) 基于 Bradbury 等人提出的并发变异算子, 运用反射、元变异、设计模式等技术开发了一个高效、可扩展的并发变异体自动生成工具 CMuJava;

(2) 以经验研究的方式评估了 CMuJava 生成并发变异体的正确性、充分性和变异体生成效率.

本文第 1 节讨论并发变异测试的基本原理. 第 2 节介绍研制的 CMuJava 的机理与关键技术. 第 3 节以经验研究的方式评估 CMuJava 生成的并发变异体的正确性、充分性和生成效率. 第 4 节介绍相关研究工作. 第 5 节总结全文.

## 1 并发变异测试

在传统变异测试中, 测试人员首先应用变异算子生成变异体集合 $\{P_1', P_2', ..., P_n'\}$, 每个变异体模拟待测程序可能存在的一个故障. 对于给定的测试用例集 $T$, 在原始程序 $P$ 和变异体 $P_i'$ $(i=1,2,...,n)$ 上执行 $T$ 中所有的测试用例. 如果存在某个测试用例 $t \in T$ 使得 $P$ 和 $P_i'$ 的输出不同, 则称变异体 $P_i'$ 被“杀死”; 如果不存在一个测试用例使得 $P$ 和 $P_i'$ 输出不同, 则称 $P_i'$ 为 $P$ 的等价变异体. 通过公式(1)计算变异得分, 评估测试用例集 $T$ 的充分性和测试技术的有效性:

$$MS(M,T)=\frac{|killed(M,T)|}{|M|-|eqv(M)|}\times 100\% \tag{1}$$

其中, $MS(M,T)$表示变异得分, $|killed(M,T)|$表示可杀死变异体的数量, $|M|$表示所有变异体的数量, $|eqv(M)|$表示等价变异体的数量.

与传统变异测试不同, 并发变异测试是对程序中特有的并发机制应用变异算子生成并发变异体, 模拟并发故障类型. 传统变异测试定义为 5 元组 $E=(P,S,D,L,A)$[19], 其中, $P$ 是原始程序; $S$ 是规格说明; $D$ 是测试用例集; $L=(l_1,l_2,\ldots,l_n)$, $l_i(i=1,2,\ldots,n)$表示 $P$ 中语句的位置; $A=(A_1,A_2,\ldots,A_n)$, $A_i(i=1,2,\ldots,n)$表示位置 $l_i$ 处可应用的变异算子集合, $|A_i|$是变异算子的数量. 通过扩展传统变异测试模型, 本文给出的并发变异的测试模型如下: $E=(P,S,D,L,C,A)$, 其中, $P$ 是原始并发程序; $S$ 是规格说明; $D$ 是测试用例集; $L=(l_1,l_2,\ldots,l_n)$, $l_i(i=1,2,\ldots,n)$表示 $P$ 中语句的位置; $C=(c_1,c_2,\ldots,c_n)$, $c_i(i=1,2,\ldots,n)$表示 $P$ 中并发机制的位置; $A=(A_1,A_2,\ldots,A_n)$, $A_i(i=1,2,\ldots,n)$是位置 $c_i$ 的可应用的并发变异算子集合, $|A_i|$是变异算子的数量.

图 1 描述了并发变异测试的流程, 主要步骤如下.

① 分析原始并发程序中所有的并发机制;

② 选择合适的并发变异算子, 生成并发变异体集合;

③ 识别等价并发变异体, 并予以删除;

④ 采用测试用例执行并发变异测试(分别执行原始程序与并发变异体程序);

⑤ 计算变异得分: 如果变异得分满足要求, 则变异测试结束; 否则, 执行步骤⑥;

⑥ 增加测试用例并添加到当前测试用例集, 执行步骤④.

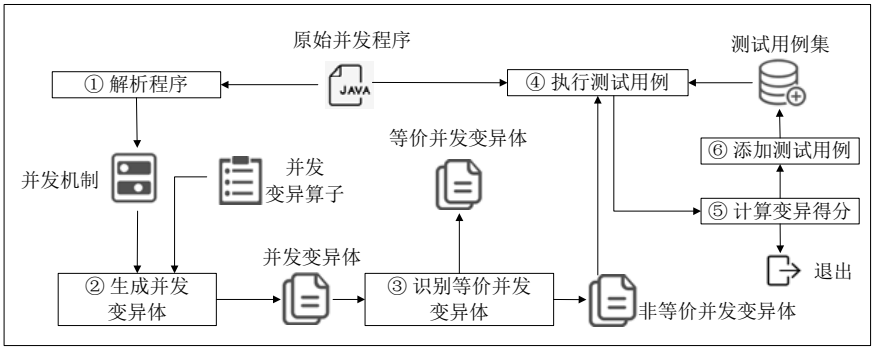


图 1 并发变异测试过程

并发变异算子是实施并发变异测试的关键, 不仅定义了并发变异体生成的语法修改规则, 还限定了变异测试所能模拟的故障类型. 依据 Java (J2SE 5.0)并发特性及 Java 并发程序的故障类型[21], Bradbury 等人提出了面向 Java 程序的并发变异算子[20](见表 1), 这些并发变异算子能够较全面地模拟常见的并发故障类型(可参考文献[20,22]进一步了解并发变异算子信息和并发变异体示例).

表 1 Java 并发变异算子

| 变异算子类别 | 简称 | 描述 |
|---|---|---|
| 修改并发方法参数 | MXT | 修改方法参数 |
| | MSP | 修改同步代码块参数 |
| | ESP | 交换同步块参数 |
| | MSF | 修改信号量公平性 |
| | MXC | 修改权限许可和线程数量 |
| | MBR | 修改同步屏障参数 |
| 修改并发方法调用 | RTXC | 删除线程方法调用 |
| | RCXC | 删除并发机制方法调用 |
| | RNA | 使用 notify()替换 notifyAll() |
| | RJS | 使用 join()替换 sleep() |
| | ELPA | 交换权限/锁获取方法 |
| | EAN | 将原子调用替换为非原子调用 |
| 修改关键字 | ASTK | 为方法添加 static 关键字 |
| | RSTK | 删除方法 static 关键字 |
| | ASK | 为方法添加 synchronized 关键字 |
| | RSK | 删除方法 synchronized 关键字 |
| | RSB | 删除 synchronized 代码块 |
| | RVK | 删除 volatile 关键字 |
| | RFU | 删除 unlock 方法所在的 finally 代码块 |
| 交换并发对象 | RXO | 替换并发机制 |
| | EELO | 交换锁对象 |
| 修改临界区 | SHCR | 移动临界区 |
| | SKCR | 收缩临界区 |
| | EXCR | 扩大临界区 |
| | SPCR | 拆分临界区 |

并发变异测试执行过程主要包括并发变异体生成和并发变异体执行两个阶段. 对于复杂的并发程序, 大量的并发变异体的生成与执行均需要消耗很长的时间. 为了提高并发变异测试执行效率, 人们开发了并发变异分析系统. Gligoric 等人提出了一个变异测试优化工具[23], 集成了 4 种并发变异体执行的优化技术, 实验结

果表明，该工具能够减少 77%的变异测试时间. Madiraju 等人[9]提出了部分变异方法，选择程序中的复杂部分或方法进行变异，大量减少并发变异体的数量，开发的支持工具仅实现了 3 种并发变异算子. Gligoric 等人[10]采用选择变异的方法提高并发变异测试效率，即：采用并发变异算子的子集生成并发变异体，减少并发变异体的数量. 已有的研究工作通过减少并发变异体数量来提高变异测试效率，但都采用手工的方法生成并发变异体. 为了提高并发变异测试的自动化程度，需要提供一个高效的并发变异测试支持系统，系统不仅支持并发变异体的高效生成，而且应具备良好的可扩展性.

## 2 CMuJava

由于并发变异算子定义了针对某种并发机制的程序变换规则，同一程序中可能应用变异算子多次，因此会生成多个并发变异体实例；另一方面，新型的并发故障需要新的并发变异算子支持. 基于 Bradbury 等人提出的并发变异算子，我们开发了一个面向 Java 程序的并发变异体自动生成工具 CMuJava.

### 2.1 CMuJava的基本原理

CMuJava 的基本原理如图 2 所示：首先，程序分析器(program analyzer)读取 Java 原始程序(original program)，采用编译时反射机制分析技术[24]生成元对象(MetaObject)，获取原始程序中包含的并发机制(concurrent mechanism)；变异体生成器(concurrent mutant generator)依据变异算子定义的规则，采用 MSG (mutant schema generation)方法[25]，高效地生成并发元变异体(concurrent metamutant)；变异体执行器(concurrent mutant executor)对原始程序和并发变异体执行测试用例，得到相应的测试结果(test result).

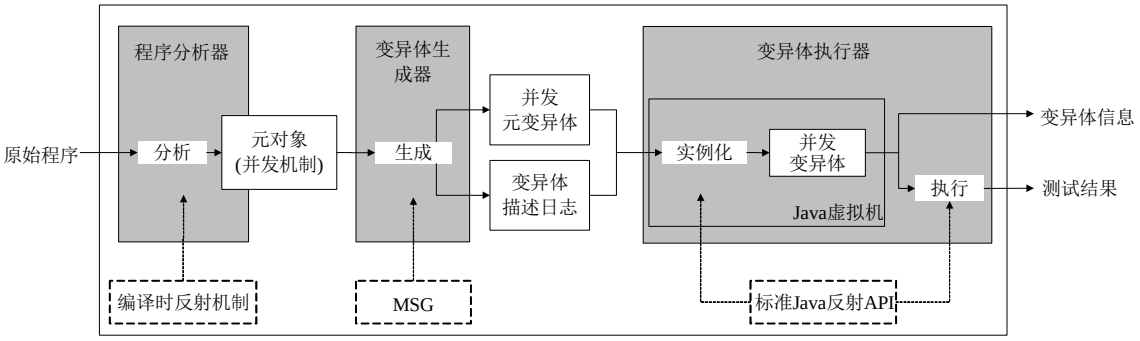


图 2 CMuJava 基本原理

### 2.2 CMuJava系统的关键技术

程序分析器对原始 Java 并发程序进行解析，并获取相应的语法信息. 针对传统的面向过程程序，语法分析器通常首先生成抽象语法树，然后修改抽象语法树生成变异体. 由于存在继承关系、属性和方法等面向对象特征，抽象语法树不能直接访问并发程序的上述新特征[26]. 针对上述问题，采用反射技术解决 Java 程序的解析问题[5]：首先提供表示某个类对象逻辑结构的引用，程序员根据该引用提取该类对象的相关信息；然后提供一个在执行过程中对程序进行修改的 API；最后，动态调用实例化对象的方法. 我们将上述反射技术应用到并发程序解析中. 具体来说：一方面, Java 语言使用专用 API 提供内置的反射功能，允许 Java 程序执行诸如请求给定对象的类，查找类中方法以及调用这些方法等功能；另一方面，这些 API 不支持更改程序行为的反射功能[5]. 为此，人们提出了一些反射系统[27−30]以弥补上述 Java 反射 API 的不足之处. 其中，OpenJava[27]是一个编译时反射系统，利用元程序处理表示程序逻辑实体的元对象，在编译时可以引用这些元对象信息. 本文基于 OpenJava 实现对 Java 原始并发程序的语法解析，将待测程序的所有信息存储到一个元对象(MetaObject)中. 元对象采用树状结构表示相应程序的语法信息和逻辑结构信息，各个节点提供了获取继承关系以及方法定义等信息的 API. 通过这些 API，可以对程序进行解析与修改，有效解决了并发程序的解析问题.

变异体生成器根据并发变异算子高效地生成并发变异体. 传统的变异体生成方法依据变异算子规则，修

改原始代码的副本独立生成变异体. 该方法不仅生成速度慢, 而且需要大量的存储空间. 为了克服上述局限性, 人们提出了一种元变异体生成方法(即 MSG[25]). MSG 方法将多个变异体合成到一个参数化程序中, 该参数化程序称为元变异体, 包含了待测程序所有变异体的信息和功能, 一个元变异体仅需一次编译. 鉴于 MSG 具有减少空间开销、不影响编译速度等优点, 本文采用 MSG 技术进行并发变异体生成, 提高并发变异测试的效率. 具体来说, CMuJava 解析得到的元对象, 查找并收集可变异点, 通过修改可变异点生成元变异体, 同时产生变异体描述日志(mutants description log), 记录每个变异点可应用的变异操作.

下面, 我们以并发变异算子 MSP (modify synchronized parameter)为例, 讨论并发元变异体的生成过程. MSP 变异算子定义的变异规则为修改 synchronized 代码块的同步参数, 即: 将“this”关键字替换为其他对象, 或者将其他对象修改为“this”关键字. 首先, 将源程序中的 synchronized 代码块抽象为 Synchronized (getParameter(orignalArg,objList)){…}; 然后, 在运行时刻通过 getParameter(·)方法动态地获取同步参数. getParameter(·)方法存在两个参数, 其中, “originalArg”表示原始同步参数, “objList”表示可以用于替换的对象列表. 由于 MSP 仅适用于 synchronized 的同步参数修改, 在应用 MSP 生成并发变异体时, 应判定当前程序是否适用 MSP 并发变异规则. 如果适用, 则首先自动获取该代码块的所有成员变量, 然后将 synchronized 代码块的原始参数和成员变量列表传递给 getParameter(·)方法, getParameter(·)方法进行同步参数替换, 则可生成待测程序的 MSP 元变异体(包含了所有 MSP 变异体的参数化程序). 图 3 示例了源程序及其 MSP 元变异体. 图 4 示例了 MSP 变异体描述日志(包括变异点和相应的变异操作).

```
1.  private Object lock1=new Object(·);
2.  private Object lock2=new Object(·);
3.  private Object lock3=new Object(·);
4.  public void methodA(·) {
5.      synchronized(lock1) {…}
6.  }
```

(a) Original code

```
1.  private Object lock1=new Object(·)
2.  private Object lock2=new Object(·)
3.  private Object lock3=new Object(·)
4.  public void methodA(·) {
5.      synchronized(lock1,lock2,lock3,this) {…}
6.  }
```

(b) MSP metamutant

图 3 源程序和 MSP 元变异体示例

```
变异点: synchronized(lock1)
变异操作:
synchronized(lock2)
synchronized(lock3)
synchronized(this)
```

图 4 MSP 变异体描述日志示例

变异体执行器将元变异体加载到 Java 虚拟机(JVM), 调用标准 Java 反射 API (standard Java reflection API), 依据变异体描述日志实例化每一个元变异体, 得到具体的并发变异体, 对原始程序和并发变异体执行 JUnit 测试脚本, 得到测试结果, 包括变异得分以及每一个变异体的“杀死”信息; 另一方面, 将变异体信息(mutant information)传递给变异体显示器(concurrent mutant viewer), 包括变异体的类型和数量以及变异的具体位置和内容.

CMuJava 目前支持 25 种并发变异算子. 为支持新的并发变异算子, 我们采用访问者设计模式(visitor)[31]提高 CMuJava 的可扩展性. 如上所述, CMuJava 解析 Java 原始程序后得到一个元对象, 然后查询元对象的各个节点, 对可变异的节点进行语法修改, 生成并发变异体. 如果将可变异节点的判断逻辑与修改逻辑都放到元对象的每个节点内部, 那么当新增一个变异算子时, 就需要对元对象内部的一个或多个节点的操作方法进行修改. 显然, 这样的设计方案易导致节点修改的代码复杂性增加, 可扩展性差. 不难看出, CMuJava 元对象的数据结构稳定, 无需新增节点或者对已有节点进行修改, 但是不同的变异算子需要对元对象进行各种操作. Visitor 设计模式适用于被访问对象结构相对稳定且操作不确定的情况. 采用 Visitor 设计模式构造 CMuJava 的类图如图 5 所示. 其中, CompilationUnit 充当结构对象的角色, 负责存储所有的节点对象, 提供遍历节点对象

的方法. 访问者类 Mutator 和 MutatorWriter 实现 ParseTreeVisitor 类声明的访问节点的 visit(·)方法. 当出现新型的并发变异算子时, 仅需增加 Mutator 与 MutatorWriter 继承类(即新的访问者类), 并重载 visit(·)方法. 例如, 文献[10]定义了一种新型并发变异算子 RTS (remove thread start)——删除调用 start(·)方法语句. 为了在 CMuJava 中支持 RTS, 首先继承图 5 中的 Mutator 与 MutatorWriter, 然后在 Mutator 子类中重载 visit(·)方法实现 RTS 的变异规则, 在 MutatorWriter 子类中重载 visit(·)方法, 实现变异体的输出. 不难看出: 通过上述类的扩展可以快速实现新定义并发变异算子, 而无需修改现有的程序代码.

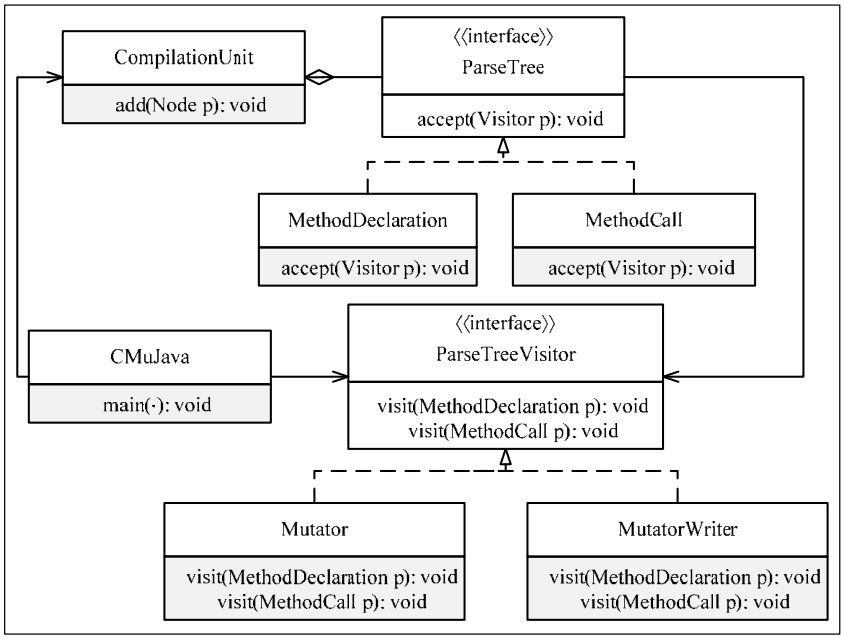


图 5 CMuJava 类图

基于 Java 语言开发了工具原型 CMuJava, 按需生成不同层次与类型的变异体, 包括传统方法层次变异体(traditional mutants)、类层次变异体(class mutants)、并发变异体(concurrent mutants). CMuJava 主要包括如下 4 个组件.

(1) 语法分析组件: 分析原始程序, 获取待测程序的并发信息;

(2) 变异体生成组件: 生成传统变异体和并发变异体;

(3) 变异体显示组件: 显示变异体数量和变异体类型, 对比变异体与源程序的差异;

(4) 变异体执行组件: 负责执行测试用例, 计算测试用例集的变异得分.

## 3 经验研究

本文选取 7 个真实的基准并发程序以验证 CMuJava 生成的并发变异体的正确性、完备性与效率.

### 3.1 研究问题

通过经验研究, 我们希望回答如下 3 个问题.

(1) CMuJava 生成的并发变异体的正确率如何?

采用一些广泛使用的并发程序评估 CMuJava 是否能够识别程序的并发机制并生成正确的并发变异体, 且与人工生成的并发变异体的正确率进行比较.

(2) CMuJava 是否能够生成完备的并发变异体集合?

首先分析并发程序内部所使用的并发机制, 确定应该生成的并发变异体的数量; 然后分析与比较 CMuJava 生成与人工生成的变异体数量, 评估 CMuJava 生成变异体的完备性.

(3) CMuJava 是否能够提高并发变异体生成的效率?

通过分析与比较 CMuJava 与人工生成并发变异体所需要的时间, 评估 CMuJava 的效率.

### 3.2 实验对象

我们选择不同规模、不同复杂度的 7 个并发程序以验证与评估本文开发的 CMuJava. 表 2 列出了实验程序的名字、代码行数以及可应用的并发变异算子. 其中, Bin、ThreadID (TID)、FineGrainedHeap (FGH) 选自文献[32]的基准并发程序, StripedSizedEpoch (SSE)、LogicalOrderingAVL (LOAVL)、TranscationalFriendly TreeSet (TFTS)来自于一个专门的基准并发程序库 Synchrobench[33], JmsManager (JM)来自于 Apache 开源库下的日志组件 Log4j (https://logging.apache.org/log4j/2.x/). 这些基准并发程序涵盖了 Java 语言的常用并发机制, 不同实验程序涉及了不同的并发机制.

表 2 实验程序信息

| 编号 | 程序名 | 代码行数 | 可应用的并发变异算子 |
|---|---|---|---|
| 1 | Bin | 42 | RSK |
| 2 | TID | 50 | ASK, RSK, RVK |
| 3 | FGH | 225 | ELPA, RCXC |
| 4 | SSE | 148 | MSP, RTXC, RNA, RSB, RVK |
| 5 | LOAVL | 1 088 | EELO, ELPA, RCXC, RVK, RXO |
| 6 | TFTS | 617 | ASK, RJS, RSK, RVK |
| 7 | JM | 490 | ASK, RVK, MSP, SHCR, SPCR SKCR, EXCR |

### 3.3 度量指标

经验研究的自变量为提出的并发变异体生成工具 CMuJava 生成的变异体和测试人员人工生成的变异体. 其中, CMuJava 为实验组, 测试人员为对照组. 两个实验组分别对基准并发程序进行变异体生成实验.

本文采用正确率($R$)指标来评估生成的并发变异体的正确性, 其计算公式如公式(2)所示:

$$R=\frac{|M_p|}{|M_t|}\times 100\% \tag{2}$$

其中, $|M_p|$是正确的并发变异体数量, $|M_t|$是所有的并发变异体数量.

通过统计生成的并发变异体的数量来评估生成并发变异体的完备性. 完备性是指实际生成的正确的并发变异体数量与理论上应该生成的并发变异体数量的比值, 比值越高, 说明完备性越强. 根据实验结果计算 CMuJava 和测试人员人工生成的所有正确的并发变异体数量, 并与待测程序理论上可能生成的并发变异体数量进行对比, 判断两种方式是否能够正确地识别出程序里所有的可变异点, 并生成相应的并发变异体.

评估效率的指标是生成并发变异体的时间(单位为秒). 分别记录 CMuJava 和测试人员从选择待测程序到生成并发变异体期间所用到的时间. 需要说明的是: 在实验开始前, 我们对测试人员进行了培训, 使其熟悉并发变异测试过程与并发变异算子规则, 这部分培训时间不计入测试人员的变异体生成时间. 实验过程中, 尽量保证测试人员不受其他因素干扰, 如存在不可避免的中断, 则从总计时中扣除相应的中断时间. 通过对比两种方式生成并发变异体时间, 评估 CMuJava 的效率.

### 3.4 实验设置

1) 实验环境

CMuJava 运行在 64 位 Windows 10 操作系统, Java 环境为 JDK 1.8, 主要配置为 Intel Core i7-7700HQ CPU, 16 GB 运行内存. 测试人员手动生成并发变异体时所使用的集成开发环境为 Eclipse.

2) 实验步骤

本文实验基本流程如下.

(1) 从实验对象中选取一个待测程序, 测试人员阅读程序, 了解程序的基本功能和程序所使用的并发机

制，根据并发变异算子寻找待测并发程序里是否存在相应的可变异点，并生成相应的变异体;

(2) CMuJava 读取待测程序并进行解析，根据并发变异算子的规则，寻找符合并发变异算子变异规则的可变异点，依据当前并发变异算子进行语法变化，生成对应的并发变异体. 重复上述过程，直至完成所有并发变异算子的变异体生成. 以待测程序 TID 为例, CMuJava 变异体生成界面如图 6 所示. 测试人员首先在左侧“File”栏选择“ThreadID.java”为待测原始程序，然后在右侧“Java Mutation Operator”栏中勾选该待测程序适用的并发变异算子，点击“Generate”按钮. CMuJava 将按照上述设置生成相应的变异体集合，并将生成的变异体集合存放于指定目录下;

(3) 计算 CMuJava 和测试人员完成该待测程序的并发变异体生成工作所使用的时间，统计 CMuJava 和测试人员生成正确并发变异体的数量.

为了保证实验结果的统计特性，测试工具实验重复了 30 次, 5 名测试人员参与了实验.

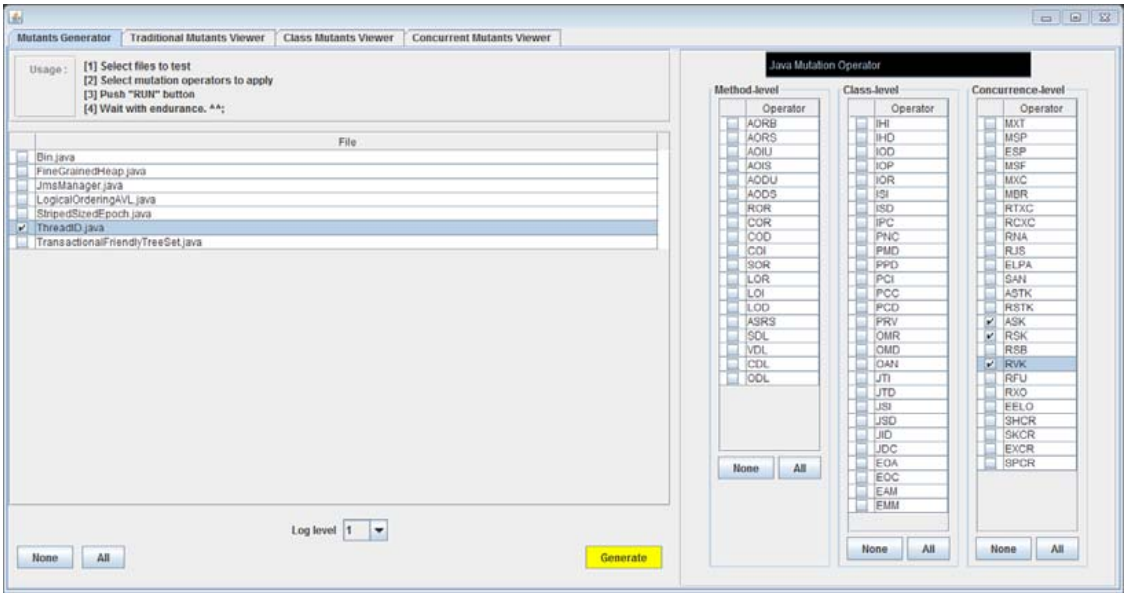


图 6 CMuJava 变异体生成界面

### 3.5 实验结果分析

(1) 并发变异体生成的正确率评估结果.

图 7 展示了不同实验对象中 CMuJava 生成与测试人员(人工生成)的并发变异体正确率的比较结果. 其中，横轴为实验次数，纵轴为实验对象，“CMuJava>测试人员”表示 CMuJava 生成并发变异体正确率高于人工生成的次数，“CMuJava=测试人员”表示两种方式生成并发变异体正确率相等的次数. 由于 CMuJava 具有语法检查功能，如果生成的变异体无法通过语法检查，则不会被输出到文件系统中. 因此，CMuJava 生成变异体的正确率均为 100%. 由于测试人员的能力差异，部分测试人员生成的并发变异体有错. 其中, TID、FGH、TFTS 这 3 个实验对象中，人工生成的并发变异体的正确率较低. 上述结果表明, CMuJava 有助于提升手工生成并发变异体的正确率.

(2) 并发变异体生成的完备性评估结果.

图 8 显示了针对不同实验对象，测试人员和 CMuJava 两种方式生成并发变异体数量平均值的对比结果. 对于简单程序(如 Bin 程序)，测试人员和 CMuJava 均能够正确识别程序中的并发机制，并生成所有的并发变异体. 随着并发程序代码复杂度与适用的并发变异算子数量的增加，人工生成方式遗漏的并发变异体数量有所增加，CMuJava 依然能够正确识别程序中所有的可变异点并生成相应的并发变异体. 上述实验结果表明，

CMuJava 可以生成完备的并发变异体集合.

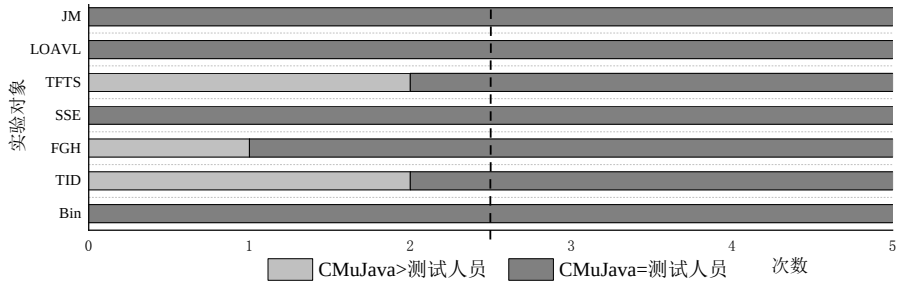


图 7　CMuJava 与人工生成并发变异体的正确率比较

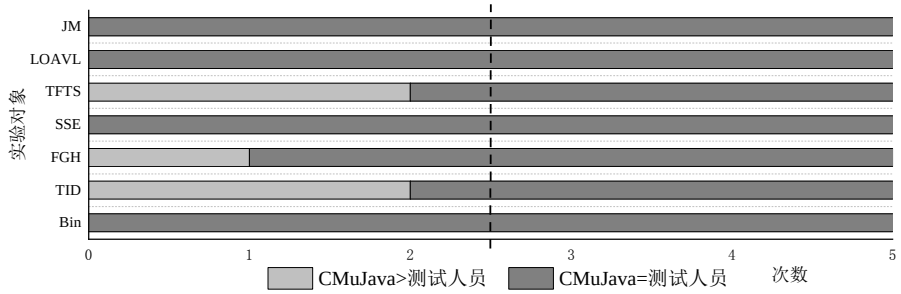


图 8　CMuJava 与人工生成的并发变异体数量比较

(3) 并发变异体生成效率评估结果.

表 3 与图 9 比较了测试人员与 CMuJava 生成并发变异体的效率.

表 3　生成并发变异体用时比较　(s)

| 实验对象 | 人工生成 | | | | | | | CMuJava 生成 | |
|---|---|---|---|---|---|---|---|---|---|
| | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $E$ | $D$ | $E$ | $D$ |
| Bin | 512 | 523 | 448 | 487 | 524 | 498.80 | 8.23E+02 | 0.01 | 1.11E−05 |
| TID | 555 | 545 | 508 | 540 | 582 | 546.00 | 5.72E+02 | 0.02 | 1.36E−05 |
| FGH | 1 004 | 944 | 900 | 893 | 987 | 945.60 | 1.20E+03 | 0.11 | 3.80E−04 |
| SSE | 1 638 | 1 701 | 1 534 | 1 758 | 1 880 | 1702.20 | 1.34E+04 | 0.14 | 4.34E−04 |
| TFTS | 2 903 | 3 251 | 2 953 | 3 361 | 3 097 | 3113.00 | 3.01E+04 | 0.68 | 5.71E−02 |
| LOAVL | 4 553 | 4 452 | 4 353 | 4 616 | 4 520 | 4498.80 | 8.11E+03 | 0.72 | 8.36E−02 |
| JM | 6 765 | 7 219 | 6 891 | 6 881 | 6 098 | 6770.80 | 1.36E+05 | 0.92 | 9.50E−02 |

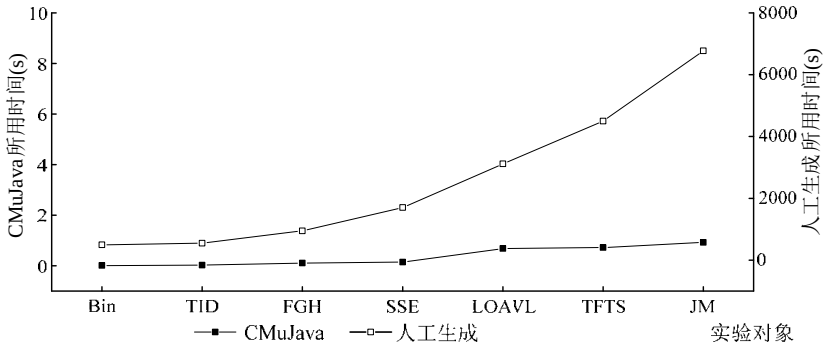


图 9　并发变异体生成用时对比图

表 3 中列出了 5 个测试人员($T_1$~$T_5$)人工生成变异体的时间、平均值($E$)及方差($D$), CMuJava 重复 30 次实

验的时间、平均值(*E*)及方差(*D*). 不难看出: (1) 随着程序规模与复杂度的增加, 生成的并发变异体数量有所增加, 测试人员所需要的时间也随之急剧增加, 但 CMuJava 在针对不同程序生成并发变异体所用时间差距上明显较小; (2) CMuJava 与手工生成并发变异体相比, 耗时差距明显, 高达数千倍. 因此, CMuJava 具有更高的效率和稳定性.

上述实验结果表明: 本文开发的 CMuJava 显著提升了并发变异测试的自动化程度, 有效提高了并发变异体生成的正确性、完备性与效率. 具体来说, 与人工生成方式相比:

(1) CMuJava 提高了并发变异体生成的正确率约 4%. 对于较为复杂的并发变异算子(例如 RCXC 和 ASK), 正确率提高得更加明显;

(2) CMuJava 多生成了 10%的并发变异体, 有效地提升了并发变异体生成的完备性, 弥补了人工生成方式存在的并发变异体遗漏问题;

(3) CMuJava 的并发变异体生成效率提高了约 8 000 倍.

### 3.6 小 结

我们选择了一组并发 Java 程序作为基线, 评估了 CMuJava 生成并发变异体时的正确性、完备性与效率. 与人工生成方式相比, CMuJava 可以更加高效地生成正确与充分的并发变异体. 与已有变异测试工具相比, CMuJava 支持较多类型的并发变异算子, 支持方法、类、并发等不同层次的变异算子, 具有良好的可扩展性. 目前, 用于评估的并发 Java 程序的规模较小, 需要通过大型且复杂的并发 Java 程序进一步验证与评估 CMuJava 的实用性.

## 4 相关工作

测试人员在变异测试领域进行了大量的研究工作. 针对 C/C++、C#、Java、SQL 等多种编程语言[34], 定义了相应的变异算子[20,21,35−39], 针对不同编程语言, 开发了变异体自动化生成工具[5,40,41], 研究了变异测试技术优化[23], 如变异算子的选择[9,10]. 下面, 给出面向 Java 程序的并发变异测试方面的相关工作介绍.

在面向 Java 程序的并发变异算子方面, Delamaro 等人根据 Java 语言的并发机制设计了 15 种并发变异算子[36], 依据变异对象的不同, 将这些并发变异算子分成了监听对象、并发方法调用、等待集合这 3 类. Farchi 等人[21]根据 Java 语言并发机制的特点, 并结合开发人员在实际编码中常犯的一些并发错误, 系统地归纳并总结出了一套并发故障模式. Bradbury 等人[20]对 Farchi 等人提出的并发故障模式进行了补充, 以这些并发故障模式为依据, 设计了包括修改并发方法参数、修改并发方法调用、修改关键字、交换并发对象和修改临界区这 5 类共 25 种并发变异算子. Wu 和 Kaiser[37]认为, 已有的并发变异算子无法生成某些并发变异体. 对已有的变异算子进行组合, 得到了 6 种新的并发变异算子, 将这些并发变异算子分成同步方法和同步代码块两类. Gligoric 等人[10]提出了 3 种新的并发变异算子, 并对 Bradbury 等人提出的 25 种并发变异算子进行了评估.

由于人工生成变异体花费大、耗费精力、易出错, 针对不同的编程语言, 人们开发出相应的变异测试支持工具[34]. 在已有的面向 Java 语言的变异测试支持工具中[5−8], MuJava 实现了传统的方法层次和类层次的变异算子[5,35], 但其并不支持并发变异体的生成. Kim 等人通过扩展 MuJava, 实现了所提出的弱变异与强变异结合的方法[16]. Javalance 为 Java 程序提供了开源的变异测试框架, 但不支持并发变异体的生成. Jumble[7]是一个面向 Java 程序字节码的变异工具, 提供了方法层次的变异算子, 并支持多线程程序的运行. 近年来, 人们开发出一些面向并发程序的变异支持工具[9,23,42]. 其中, Paraμ[9]是一个支持并发变异体生成的工具, 仅实现了 2 个类级别变异算子和 3 个并发变异算子; MutMut[23]是一个面向并发程序的优化变异执行工具, 可以节省高达 77%的变异测试时间, 但 MutMut 并不支持变异体生成; Comutation[10]实现了面向 Java 程序的 28 种并发变异算子, 但不支持开源使用.

我们的前期工作研究了多种变异测试优化技术与支持工具[14,15,43−47]: 针对真实软件故障的“群束”特征, 提出了一种分布感知的变异测试技术[14], 在扩展 MuJava 的基础上, 开发了支持非均匀分布的变异生成系统 MuJavaX[15]; 利用程序的结构信息, 提出了一种路径感知的变异体精简方法[43]; 提出了冗余变异体的概念,

开发了一种基于数据流分析的冗余变异体识别方法[44]; 利用程序合成与并发机制等技术, 开发了一种变异测试加速执行方法[45]; 针对新型服务组装程序, 提出了面向 BPEL 的变异测试框架与优化技术[46,47]. 基于上述研究积累, 本文研究了面向并发程序的变异测试技术, 以开发相应的支持系统.

## 5 总 结

由于缺少面向 Java 程序的开源并发变异测试工具, 人们通常采用手工方式生成并发变异体. 本文在面向并发程序的变异测试框架的基础上, 开发了一个面向 Java 程序的并发变异体自动生成工具 CMuJava. 采用一组并发程序集评估了 CMuJava 的有效性与性能. 实验结果表明, 使用 CMuJava 可以显著提高并发变异体生成的正确性、完备性和效率. 本文开发的 CMuJava 可以有效地支持并发变异测试的主要活动, 极大提高了并发变异测试的自动化程度与效率. 相关研究结果为评估并发程序的测试用例集的充分性与面向并发程序的测试技术有效性提供了一种手段.

进一步的研究工作包括: (1) 采用更多的并发程序来验证和评估 CMuJava 的实用性, 进一步提升 CMuJava 的可用性; (2) 研究新型服务组装程序中的并发机制, 并研制相应的并发变异测试技术与支持工具.

**References**:

[1] Bianchi FA, Margara A, Pezze M. A survey of recent trends in testing concurrent software systems. IEEE Trans. on Software Engineering, 2018, 44(8): 747−783.

[2] Demillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. Computer, 1978, 11(4): 34−41.

[3] Chen X, Gu Q. Mutation testing: Principal, optimization and application. Journal of Frontiers of Computer Science and Technology, 2012, 6(12): 1057−1075 (in Chinese with English abstract).

[4] Carver R. Mutation-based testing of concurrent programs. In: Proc. of the 1993 IEEE Int'l Test Conf. (ITC 1993). IEEE Computer Society, 1993. 845−853.

[5] Ma YS, Offutt AJ, Kwon YR. MuJava: An automated class mutation system. Software Testing Verification and Reliability, 2005, 15(2): 97−133.

[6] Schuler D, Zeller A. Javalanche: Efficient mutation testing for Java. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on Foundations of Software Engineering (ESEC/FSE 2009). ACM, 2009. 297−298.

[7] Irvine SA, Pavlinic T, Trigg L, Cleary JG, Inglis S, Utting M. Jumble Java byte code to measure the effectiveness of unit tests. In: Proc. of the 3rd Workshop on Mutation Analysis (Mutation 2007). IEEE Computer Society, 2007. 169−175.

[8] Rene J, Schweiggert F, Kapfhammer GM. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2011). IEEE Computer Society, 2011. 612−615.

[9] Madiraju P, Namin AS. Paraμ: A partial and higher-order mutation tool with concurrency operators. In: Proc. of the 4th IEEE Int'l Conf. on Software Testing, Verification and Validation Workshops (ICSTW 2011). IEEE Computer Society, 2011. 351−356.

[10] Gligoric M, Zhang L, Pereira C, Pokam G. Selective mutation testing for concurrent code. In: Proc. of the 2013 Int'l Symp. on Software Testing and Analysis (ISSTA 2013). ACM, 2013. 224−234.

[11] Kintis M, Papadakis M, Papadopoulos A, Valvis E, Malevris N. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In: Proc. of the 16th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM 2016). IEEE, 2016. 147−156.

[12] Saifan AA, Alazzam I, Akour M, Hanandeh F. Regression test-selection technique using component model based modification: Code to test traceability. Int'l Journal of Advanced Computer Science and Applications, 2016, 7(4): 90−95.

[13] Masood A, Bhatti R, Ghafoor A, Mathur AP. Scalable and effective test generation for role-based access control systems. IEEE Trans. on Software Engineering, 2009, 35(5): 654−668.

[14] Sun CA, Wang G, Cai KY, Chen TY. Distribution-aware mutation analysis. In: Proc. of the 36th IEEE Annual Computer Software and Applications Conf. Workshops (COMPSACW 2012). IEEE Computer Society, 2012. 170−175.

[15] Sun CA, Wang G. MuJavaX: A distribution-aware mutation generation system for Java. Journal of Computer Research and Development, 2014, 51(4): 874−881 (in Chinese with English abstract).

[16] Kim SW, Ma YS, Kwon YR. Combining weak and strong mutation for a noninterpretive Java mutation system. Software Testing, Verification and Reliability. 2013, 23(8): 647−668.

[17] Lipton RJ. Reduction: A method of proving properties of parallel programs. Communications of the ACM, 1975, 18(12): 717−721.

[18] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558−565.

[19] Morell LJ. A theory of fault based testing. IEEE Trans. on Software Engineering, 1990, 16(8): 844−857.

[20] Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrent Java (J2SE 5. 0). In: Proc. of the 2nd Workshop on Mutation Analysis (MUTATION 2006). IEEE Computer Society, 2006. 83−92.

[21] Farchi E, Nir Y, Ur S. Concurrent bug patterns and how to test them. In: Proc. of the 17th Int'l Symp. on Parallel and Distributed Processing (IPDPS 2003). IEEE Computer Society, 2003. 286−292.

[22] Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrent Java (J2SE 5.0). Technical Report, 2006-520, Queen's University, 2006.

[23] Gligoric M, Jagannath V, Marinov D. MutMut: Efficient exploration for mutation testing of multithreaded code. In: Proc. of the 3rd Int'l Conf. on Software Testing, Verification and Validation. IEEE Computer Society, 2010. 55−64.

[24] Kiczales G, Rivieres JD, Bobrow DG. The Art of the Metaobject Protocol. MIT Press, 1991.

[25] Untch RH, Offutt AJ, Harrold MJ. Mutation analysis using mutant schemata. In: Proc. of the 1993 Int'l Symp. on Software Testing and Analysis (ISSTA 1993). ACM, 1993. 139−148.

[26] Chevalley P. Applying mutation analysis for object-oriented programs using a reflective approach. In: Proc. of the 8th Asia-Pacific Software Engineering Conf. (APSEC 2001). IEEE Computer Society, 2001. 267−270.

[27] Tatsubori M, Chiba S, Killijian MO, Itano K. OpenJava: A class-based macro system for Java. In: Proc. of the 1st Workshop on Reflection and Software Engineering (OORaSE 1999). Springer-Verlag, 1999. 117−133.

[28] Chiba S. Load-time structural reflection in Java. In: Proc. of the 14th European Conf. on Object-Oriented Programming (ECOOP 2000). Springer-Verlag, 2000. 313−336.

[29] Kleinoder J, Golm M. MetaJava: An efficient run-time Meta architecture for Java. In: Proc. of the 5th Int'l Workshop on Object-orientation in Operating Systems (IWOOOS 1996). IEEE Computer Society, 1996. 54−61.

[30] Welch I, Stroud RJ. Kava: Using byte code rewriting to add behavioral reflection to Java. Technical Report, 704, University of Newcastle upon Tyne, 2000.

[31] Pati T, Hill JH. A survey report of enhancements to the visitor software design pattern. Software: Practice and Experience, 2014, 44(6): 699−733.

[32] Herlihy M, Shavit N. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers, 2012.

[33] Gramoli V. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: Proc. of the 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2015). ACM, 2015. 1−10.

[34] Papadakis M, Kintis M, Zhang JM, Jia Y, Traon YL, Harman M. Mutation testing advances: An analysis and survey. Advances in Computer, 2019, 112: 275−378.

[35] Offutt J, Ma YS, Kwon YR. The class-level mutants of MuJava. In: Proc. of the 2006 Int'l Workshop on Automation of Software Test (AST 2006). ACM, 2006. 78−84.

[36] Delamaro M, Pezzè M, Vincenzi AMR, Maldonado JC. Mutant operators for testing concurrent Java programs. In: Proc. of the Brazilian Symp. on Software Engineering (SBES 2001). IEEE Computer Society, 2001. 272−285.

[37] Wu L, Kaiser G. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. In: Proc. of the 23rd Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE 2011). 2011. 244−249.

[38] Ma YS, Kwon YR, Offutt J. Inter-class mutation operators for Java. In: Proc. of the 13th Int'l Symp. on Software Reliability Engineering (ISSRE 2002). IEEE Computer Society, 2002. 352−366.

[39] Wu YB, Guo JX, Li Z, Zhao RL. Mutation strategy based on concurrent program data racing fault. Journal of Computer Application, 2016(11): 3170−3177, 3195 (in Chinese with English abstract).

[40] Delgado-Pérez P, Medina-Bulo I, Palomo-Lozano F, García-Domínguez A, Domínguez-Jiménez JJ. Assessment of class mutation operators for C++ with the MuCPP mutation system. Information and Software Technology, 2017, 81: 169−184.

[41] Kusano M, Wang C. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2013). IEEE Computer Society, 2013. 722−725.

[42] Bradbury JS, Cordy JR, Dingel J. Comparative assessment of testing and model checking using program mutation. In: Proc. of the 3rd Workshop on Mutation Analysis (Mutation 2007). IEEE Computer Society, 2007. 210−219.

[43] Sun CA, Xue FF, Liu H, Zhang XY. A path-aware approach to mutant reduction in mutation testing. Information and Software Technology, 2017, 81: 65−81.

[44] Sun CA, Guo XL, Zhang XY, Chen TY. A data flow analysis based redundant mutants identification technique. Chinese Journal of Computers, 2019, 42(1): 44−60 (in Chinese with English abstract).

[45] Sun CA, Jia JT, Liu H, Zhang XY. A lightweight program dependence based approach to concurrent mutation analysis. In: Proc. of the 42nd IEEE Int'l Computer Conf. on Computers, Software, and Applications (COMPSAC 2018). IEEE Computer Society, 2018. 116−125.

[46] Sun CA, Pan L, Wang QL, Liu H, Zhang XY. An empirical study on mutation testing of WS-BPEL programs. The Computer Journal, 2017, 60(1): 143−158.

[47] Sun CA, Wang Z, Pan L. Optimized mutation testing techniques for WS-BPEL programs. Journal of Computer Research and Development, 2019, 56(4): 895−905 (in Chinese with English abstract).

**附中文参考文献:**

[3] 陈翔, 顾庆. 变异测试: 原理、优化和应用. 计算机科学与探索, 2012, 6(12): 1057−1075.

[15] 孙昌爱, 王冠. MuJavaX: 一个支持非均匀分布的变异生成系统. 计算机研究与发展, 2014, 51(4): 874−881.

[39] 吴俞伯, 郭俊霞, 李征, 赵瑞莲. 基于并发程序数据竞争故障的变异策略. 计算机应用, 2016, 36(11): 3170−3177, 3195.

[44] 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技术. 计算机学报, 2019, 42(1): 44−60.

[47] 孙昌爱, 王真, 潘琳. 面向 WS-BPEL 程序的变异测试优化技术. 计算机研究与发展, 2019, 56(4): 895−905.

**孙昌爱**(1974－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件测试, 故障定位, 服务计算.

**代贺鹏**(1993－), 男, 硕士, 主要研究领域为软件测试.

**耿宁**(1996－), 女, 学士, 主要研究领域为软件测试.

**顾友达**(1995－), 男, 学士, 主要研究领域为软件测试.