摘 要: 并发系统是现在普遍存在的计算机软件系统.软件测试是一种重要的、广泛使用的软件质量保证手段,可以发现软件中存在的缺陷和故障.并发变异测试是一种基于并发故障的软件测试技术,根据并发变异算子改变待测程序的语法结构,生成并发变异体.该技术可以评估测试用例集的充分性和测试技术的有效性.目前开源的面向Java程序的变异测试工具不支持并发变异体生成,而人工生成并发变异体需要大量的测试资源.本文依据Java并发变异算子,设计并实现了一个并发变异体自动生成工具CMuJava,采用经验研究验证了CMuJava生成并发变异体的正确性、完备性及效率.

关键词: 并发变异测试; 并发变异算子; 并发变异体; 测试工具

**Abstract**: Concurrent system is a common computer software system. Software testing is an important and widely used mean to ensure software quality, which can find defects and faults in software. Concurrent mutation testing is a software testing technology based on concurrent faults, which changes the grammar structure of the program under test according to the concurrent mutation operator and generates concurrent mutation. This technique can evaluate the adequacy of test case sets and the effectiveness of test techniques. So far, exsiting mutation testing tools for Java programs do not support the generation of concurrent mutants, and manual generation of concurrent mutants requires a large amount of test resources. In this paper, we have designed and implemented an automatic generation tool, CMuJava, for generating concurrent mutants, and conducted an empirical study to evaluate the correctness, completeness and efficiency of the concurrent mutants generated by CMuJava.

**Key words**: concurrent mutation testing; concurrent mutation operator; concurrent mutant; testing tools

并发系统[21]包含多个执行流,这些执行流可以同时运行并且彼此可以交互.并发系统在一些应用领域已经被广泛应用,典型的有多核计算、为单个节点开发应用程序[1]等.并发程序中存在的多个并发的执行流的执行次序往往是不确定的,执行流之间不确定的相互作用和影响容易引发程序中出现故障,如何有效的检测并发程序中的故障,提高并发程序的可靠性至关重要.

软件测试[2]是一种广泛采用的软件质量保证手段,其中测试用例的选择是软件测试中的关键问题.变异测试[3]是一种基于故障的软件测试技术,该技术可以用来评估测试用例集的充分性.从一定角度对测试充分性进行了度量.测试人员首先分析待测程序设计出一系列变异算子[14],然后对待测程序应用变异算子产生大量的变异体[14],在识别出等价变异体[14]后,若已有测试用例不能杀死所有非等价变异体,则需要设计新的测试用例以提高测试充分性.

Carver将变异测试理论和技术应用到针对并发程序的软件测试,提出了并发变异测试[4], 通过应用并发变异算子改变待测程序的语法结构,生成并发变异体.目前并发变异测试相关研究正处于起步阶段,主要研究内容集中于并发变异算子方面[5].并发变异算子作为并发变异测试的基础,它定义了对待测并发程序进行语法修改的规则,在并发变异测试中有着极其重要的作用.另外,目前已有的开源的面向Java语言的变异测试工具（例如MuJava[10],MuClipse[11]和Javalance[12]等）不支持生成并发变异体.研究人员通常采用耗时的人工方式来生成并发变异体,设计并实现一个能够自动生成并发变异体的工具成为一个亟待解决的问题.

为了解决面向Java程序的并发变异测试缺乏开源工具支持的问题, 本文根据Java并发变异算子,设计并实现了一个自动生成并发变异体的工具CMuJava,提高了面向Java程序的并发变异测试技术的自动化程度.本文的主要贡献有：

1. 基于Brabury等人提出的25种并发变异算子[8]实现了一个自动生成并发变异体的工具CMuJava.
2. 扩展了面向Java程序的变异性测试工具MuJava的功能.
3. 以经验研究的方式验证了工具的正确性、完备性和效率.

本文第一部分对研究工作的相关背景进行介绍,第二部分介绍工具的设计与实现,第三部分以经验研究的方式验证工具的正确性、完备性和效率,第四部分列出实验结果并讨论,第五部分介绍相关研究工作,第六部分总结与展望.

# 研究背景

本节简单介绍软件测试、变异测试和并发变异测试方面的背景知识.

## 变异测试

变异测试[3]是一种可以用来定量评估测试用例集充分性的技术,通过应用变异算子改变待测程序的语法结构,生成与待测程序有语法差异的变异体来模拟待测程序可能存在的故障.

变异测试的基本流程如图1所示,具体步骤如下：

1. 应用变异算子改变待测程序的语法结构,生成变异体.
2. 从生成的大量变异体中识别出等价变异体,剩下的部分则为非等价变异体.
3. 在非等价变异体上执行测试用例,如果测试用例集中的测试用例能够“杀死”所有的非等价变异体，则变异测试结束.否则还需要继续设计测试用例并添加到当前测试用例集中,直到当前测试用例集中的测试用例能够“杀死”所有的非等价变异体.



Fig.1 Mutation Testing process

图1 变异测试流程

下面对于上述流程中所涉及到的一些概念进行解释.

变异算子[14]:变异算子是一条规则,在符合语法的前提下,它定义了对待测程序进行语法变化生成变异体的变化规则.图2给出了一个变异算子示例,该变异算子所定义的规则是将逻辑运算符<改变为<=.语句i<100在经过该变异算子的作用后变成了i<=100并生成了相应的变异体.

|  |  |
| --- | --- |
| 程序p | 变异体q |
| …  for (int i = 0; i<100; i++){  …  }  … | …  for (int i = 0; i<=100; i++){  …  }  … |

Fig.2 Example of mutation operator

图2 变异算子示例

可杀除变异体[14]:如果测试用例集T中存在测试用例t,在变异体q和原程序p上的执行结果不一致,则称该变异体相对于测试用例集T是可杀死变异体.

可存活变异体[14]:如果测试用例集T中不存在测试用例t,在变异体q和原程序p上的执行结果不一致,则称该变异体相对于测试用例集T是可存活变异体.部分可存活变异体可通过设计新的测试用例“杀死”,转化为可杀死变异体.

等价变异体[14]:如果待测程序和变异体之间仅有语法差异,但是所表达的语义完全相同,即所有的测试用例在待测程序p和变异体q上执行的结果完全一致,则称变异体q是待测程序p的等价变异体.图3给出了一个等价变异体的实际例子,该变异算子将待测程序中的逻辑运算符<变换成了!=.虽然变异体和原程序之间有了语法差异,只要循环体内不存在对变量i进行修改的语句.则程序的执行结果就是完全一致的,也就是变异体q是原程序p的等价变异体.

|  |  |
| --- | --- |
| 程序p | 变异体q |
| …  int sum = 0;  for (int i = 0; i<100; i++){  sum += i;  }  … | …  int sum = 0;  for (int i = 0; i != 100; i++){  sum += i;  }  … |

Fig.3 Example of equivalent mutant

图3 等价变异体示例

变异测试最终通过变异评分[14]来定量评估测试用例集设计的充分性,变异评分的计算方式如式1所示：

MS(M, T) = killed(M,T) ÷ {|M| - eqv(M)} (1)

MS(M,T)表示变异评分,killed(M,T)表示可杀死变异体数量,|M|表示在变异算子的作用下生成的所有的变异体数量,eqv(M)表示已识别的等价变异体数量.从表达式含义可以看出MS(M,T)取值范围为[0,1],取值越接近1表明测试用例集的充分性越高.由于针对特定程序,变异体总数量和等价变异体的数量是固定的,所以要想提高变异得分,就需要提高测试用例集所能识别的变异体数量,即提高测试用例集的充分性.

变异测试除了可以用于评估测试用例集的充分性,也可以用于模拟待测程序的真实故障,辅助评估研究人员所提出的测试方法和技术的有效性[14].

## 并发变异测试

并发程序中存在多个并发执行的线程,线程之间通常显式或隐式地共享一些存储空间.线程间执行顺序的不确定性使得并发程序的设计、实现、测试和修复更加困难.由于并发程序执行顺序的不确定性,揭示并发故障不仅需要执行具有揭示故障能力的测试用例,还需要线程之间的交错符合某种特定的模式.如何保证并发软件的质量成为一个受到广泛关注的开放问题.

并发变异测试[4]是一种基于并发故障的软件测试技术.首先根据待测并发程序所运用的并发机制选择合适的并发变异算子,然后在待测并发程序上应用并发变异算子可生成相应的并发变异体,在识别出等价并发变异体后,在非等价并发变异体上执行测试用例并根据并发变异体被识别的情况计算变异评分.如果变异评分不等于1,即测试用例无法识别出所有的非等价并发变异体,则需要设计新的测试用例并添加到测试用例集中,以提高测试用例集的故障检测能力.

并发变异测试与传统的变异测试的一个重要区别在于所运用的变异算子,由于传统的方法级别变异算子和类级别变异算子无法直接导致并发故障,所以需要设计能够直接导致并发故障的并发变异算子.

Brabury等人[8]提出的面向Java程序的25种并发变异算子能够反应真实的故障,具有代表性和良好的覆盖性. Brabury等人将25种并发变异算子分成了5类:修改并发方法参数、修改并发方法调用(删除,替换和交换)、修改关键字(添加和删除)、交换并发对象和修改临界区(移动,扩展,收缩和拆分) [8],我们将在本文第2部分详细介绍.

# 面向Java程序的并发变异体生成工具-CMuJava

本节首先列出支持工具实现的25种并发变异算子的规则定义,然后介绍本文提出的支持工具的关键技术，最后介绍本文提出的面向Java程序的并发变异体生成工具.

## 并发变异算子简介

Brabury等人提出的25种并发变异算子[10]详见表4.25种并发变异算子共分为5类.

**Table 4** concurrent mutation operators for java

表4 java并发变异算子

| 变异算子类别 | 并发变异算子 |
| --- | --- |
| 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 |

1. 修改并发方法参数

该类并发变异算子的定义是修改线程和并发类的方法参数,类似于一些修改操作数的方法级别的变异算子.具体包括6种变异算子:

MXT(修改方法超时参数):应用于包含可选超时参数的wait(),sleep()和join()方法调用.

MSP(修改synchronized代码块参数): 如果关键字this或一个对象用作同步块的参数,用另一个对象或关键字this来替换参数.

ESP(交换synchronized代码块参数):如果一个临界区被多个锁保护,交换相邻的另个锁对象.

MSF(修改信号量公平性):在参数为布尔变量处,取与之相反的值.

MXC(修改并发机制初始化数量):对参数数量做i++或i--操作.

MBR(修改同步屏障参数):如果可选线程CyclicBarrier参数存在,通过删除它来修改可运行线程参数.

1. 修改并发方法调用

该类变异算子的定义是修改对并发机制的类和线程方法的调用,包括删除,替换和交换.具体包括6中变异算子:

RTXC(删除线程方法调用):删除对wait(),join(),sleep(),yield(),notify()和notifyAll()方法的调用.其中, 删除wait()方法可能会导致潜在的冲突,删除join()和sleep()方法可能会导致sleep()故障模式,删除notify()和notifyAll()方法调用会造成丢失通知故障.

RCXC(删除同步机制方法调用):删除对Lock（lock()，unlock()），Condition（signal()，signalAll()），Semaphore（acquire()，release()），Latch（countDown()和ExecutorService（例如，submit()）并发机制的调用.

RNA(使用notify()替换notifyAll()):将notifyAll()方法替换为notify(),导致notify()故障.

RJS(使用sleep()替换join()):将join()方法替换为sleep(),导致sleep()故障.

ELPA(交换权限/锁获取方法):交换权限获取方法acquire(),acquireUninterruptibly(),tryAcquire()及锁获取方法lock(),lockInterruptibly(),tryLock()的调用.

SAN(将原子调用替换为非原子调用):例如对原子变量类中的getAndSet()方法的调用被对get()方法调用和对set()方法调用所取代.

1. 修改关键字

该类变异算子是对源程序添加和删除static,synchronized,volatile和final等关键字.具体包括:ASTK、RSTK、ASK、RSK、RSB、RVK、RFU6种变异算子.

1. 交换并发对象

该类变异算子定义在存在相同并发类型的多个实例的情况下,用另一个并发实例对象替换当前对象.具体包括RXO和EELO两种变异算子

1. 修改临界区

该类变异算子的定义是通过移动、扩大、缩小、拆分来修改临界区域.具体包括4种变异算子:

SHCR(移动临界区):通过向上或向下移动临界区域,由于不再同步访问共享变量引起故障.

EXCR(扩大临界区):使临界区包括在临界区域的语句之上和之下的语句,可能会由于不必要地降低并发度而导致性能问题.

SKCR(缩小临界区):缩小临界区由于不再同步访问共享变量引起故障.

SPCR(拆分临界区):将一个临界区分成两个临界区,可能导致原子操作变为非原子操作.

## 关键技术

我们根据并发变异算子定义的规则生成变异体.首先需要对待测程序进行解析获取待测程序的语法信息,然后根据变异算子定义的规则对待测程序进行语法改变.由于传统的抽象语法树只能够表达程序的语法信息,无法表达程序的逻辑结构信息.本文使用了OpenJava[16]对待测并发程序进行语法解析,解析完成后会将待测程序的所有信息存储到一个元对象中[16].

…..

## 支持工具:CMuJava

为了提高面向Java程序的并发变异测试自动化程度,本文根据Bradbury等人提出的Java并发变异算子[8],设计并实现了并发变异体自动生成工具CMuJava.该工具扩展了Mujava[10]的功能,在能生成传统的方法级别和类级别的变异体[17][18]基础上,能生成并发变异体.

…..

# 经验研究

本文选取7个真实的并发测试基准程序验证所提工具CMujava的性能.

## 研究问题

通过实例研究,我们希望验证如下3个方面:

1. CMujava生成并发变异体的正确性:通过使用一些被广泛使用的并发程序来进行实验,验证CMuJava是否能够识别程序内部所使用的并发机制并生成正确的并发变异体,然后将实验结果和人工生成的方式进行对比分析,判断使用工具是否能够提高生成的并发变异体的正确率.
2. CMujava生成并发变异体的完备性:根据实验结果计算工具生成的所有正确的并发变异体数量,并与待测程序可能生成的理论并发变异体数量进行对比,判断工具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.然后将工具生成的变异体和人工生成的变异体数量进行对比,验证使用工具是否能够提高生成并发变异体数量的完备性.
3. CMujava生成并发变异体的效率:通过记录工具解析待测程序和生成并发变异体的时间,分析工具在对不同规模的程序生成变异体的性能变化.然后对比人工对相同程序进行并发变异体生成实验所需要的时间,分析工具对于提升生成并发变异体的效率的作用.

## 实验对象

为了让程序的规模尽可能的与现实情况比较接近,我们选取了覆盖了从几十行到上千行的不同复杂度的并发程序.选取的7个并发测试基准程序的详细信息如表1所示.程序1~3为课题组目前正在使用的并发测试基准程序,程序4~6来自于一个专门的并发测试基准程序库Synchrobench[19],程序7来自于Apache开源库下的日志组件Log4j[20].

**Table 1** The information of experimental object

表1 实验对象信息

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 程序名 | 代码行数 | 可应用的变异算子 |
| 1 | Bin | 42 | RSK |
| 2 | ThreadID | 50 | ASK, RSK, RVK |
| 3 | FineGrainedHeap | 225 | ELPA, RCXC |
| 4 | StripedSizedEpoch | 148 | MSP, RTXC, RNA, RSB, RVK |
| 5 | LogicalOrderingAVL | 1088 | EELO, ELPA, RCXC, RVK, RXO |
| 6 | TranscationalFriendlyTreeSet | 617 | ASK, RJS, RSK, RVK |
| 7 | JmsManager | 490 | ASK, RVK, MSP, SHCR, SPCR SKCR, EXCR |

为了更好地模拟真实情况下的程序并发故障,本次的实验对象涵盖了Java语言的常用并发机制,不同的并发程序使用的并发机制也不完全相同.

## 实验过程

针对前文提出的研究问题,本文设计了两组对比实验来进行试验评估.其中CMuJava为实验组,测试人员为对照组.两实验组分别对7个程序进行变异体生成实验.实验基本流程如下:

（1）从实验对象中选取一个待测程序,然后开始计时.

（2）测试人员需要先阅读程序,了解程序的基本功能和程序所使用的并发机制.CMuJava需要读取待测程序并进行解析,生成待测程序的元对象.

（3）测试人员需要理解并发变异算子所定义的规则和要求,然后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.CMuJava也需要根据并发变异算子的规则对待测程序所生成的元对象进行遍历,寻找符合并发变异算子变异规则的可变异点.如果找到相应的可变异点则进行第4步.否则继续根据下一个并发变异算子的规则进行可变异点查找直至完成所有并发变异算子的判断.

（4）按照并发变异算子所定义的规则对识别的可变异点进行语法变化生成对应的并发变异体.

（5）判断是否已生成当前程序的所有并发变异体,如果没有则继续生成下一个并发变异体.如果生成完毕则进入下一步.

（6）停止计时,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,同时分别计算CMuJava和测试人员生成并发变异体的数量和正确率.

（7）判断是否所有的待测程序都已经完成生成并发变异体的工作,如果没有完成则回到步骤1,继续选取下一个待测程序进行实验.如果已经完成,则分别计算CMuJava和测试人员所使用的平均时间,生成的并发变异体的平均正确率和完备性.

在完成以上所有步骤后可以得到测试人员和CMuJava的实验数据,然后对实验数据进行对比分析,验证使用CMuJava对于提升生成并发变异体的正确率、完备性和效率的作用.

# 实验结果与分析

## 结果

表2列出了测试人员和CMuJava分别对7个程序生成并发变异体的正确率数据.表3总结了两种方式生成并发变异体的数量.

从表2的数据中可以看出,由于不同测试人员的能力差异和对变异算子理解的差异,所以在部分程序上正确率出现了差异. 由于测试人员是在Eclipse下进行变异体生成工作,部分测试人员在生成了一些语法错误的变异体后,但是由于开发工具的语法检查,测试人员并没有输出错误的变异体,所以整体的正确率都比较高.目前几个程序产生的错误变异体都是RCXC（删除并发方法调用）变异体,由于同一个程序中存在一些方法名和返回值都相同的方法调用,但是这些方法调用并不符合变异算子的规则,所以测试人员生成了这些错误的变异体.而工具每次运行产生的变异体都相同,所以表中只列出了一次的正确率. 为了保证CMuJava生成的变异体不存在语法错误导致的无法运行情况,CMuJava在设计的时候添加了语法检查功能,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.所以CMuJava生成变异体的正确率都为100%.

**Table 2** Correct rate of concurrent mutants

表2 生成并发变异体正确率

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 变异体生成方式 | 程序1 | 程序2 | 程序3 | 程序4 | 程序5 | 程序6 | 程序7 |
| tester1 | 100% | 80% | 67% | 100% | 100% | 100% | 100% |
| tester2 | 100% | 60% | 100% | 100% | 100% | 100% | 100% |
| tester3 | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| tester4 | 100% | 100% | 100% | 100% | 93% | 100% | 100% |
| tester5 | 100% | 100% | 100% | 100% | 87% | 100% | 100% |
| CMujava | 100% | 100% | 100% | 100% | 100% | 100% | 100% |

图是测试人员和CMuJava生成变异体的正确率对比图.横轴表示两种方式下7个实验对象,纵轴表示生成变异体的正确率.从图中可以看出,使用工具对提升生成并发变异体的正确率有一定的帮助.尤其是针对一些比较特殊的并发程序的时候,正确率的提高程度更为显著.

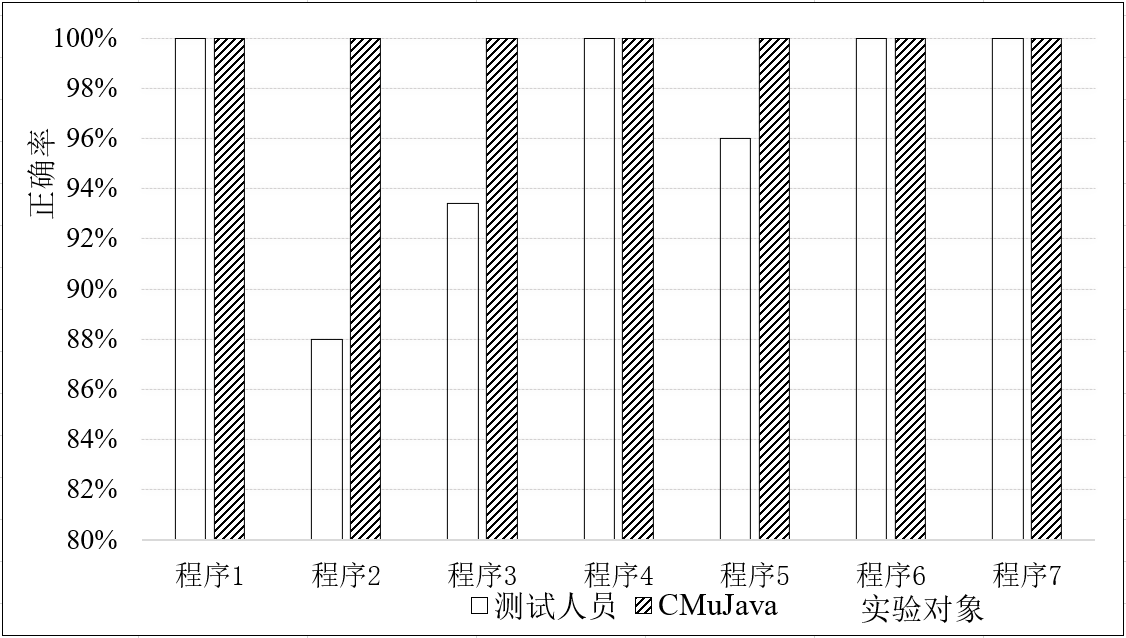


Fig. Correct rate comparison chart of concurrent mutants

图 并发变异体正确率对比图

从表3中可以看出,针对第一个比较简单的并发程序所有的测试人员和CMuJava都能够正确识别里面的并发机制并生成所有的并发变异体,但是随着待测并发程序代码复杂度和可应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也越来越多.图 是测试人员和CMuJava生成并发变异体的数量对比图.

**Table 3** Number of concurrent mutants

表3 生成并发变异体数量(个)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 变异体生成方式 | 程序1 | 程序2 | 程序3 | 程序4 | 程序5 | 程序6 | 程序7 |
| tester1 | 3 | 4 | 10 | 14 | 22 | 35 | 46 |
| tester2 | 3 | 3 | 10 | 17 | 24 | 29 | 53 |
| tester3 | 3 | 5 | 12 | 17 | 24 | 39 | 55 |
| tester4 | 3 | 5 | 12 | 12 | 27 | 43 | 55 |
| tester5 | 3 | 5 | 12 | 16 | 27 | 40 | 59 |
| CMujava | 3 | 5 | 12 | 17 | 27 | 43 | 61 |

由于生成并发变异体是一个比较枯燥和耗时的工作,主要就是根据并发变异算子规则识别可变异点,然后对可变异点进行微小的语法更改生成并发变异体.当一个可变异点存在多种变异方案或者待测程序存在较多可变异点时,测试人员会逐渐失去耐心,所以随着时间和可生成的并发变异体数量的增加,测试人员难免会少考虑一些情况,导致遗漏的并发变异体数量逐渐增加.但是工具并不会随着时间和变异体数量的增加而失去耐心导致遗漏,所以使用工具就可以很好地避免这个问题.

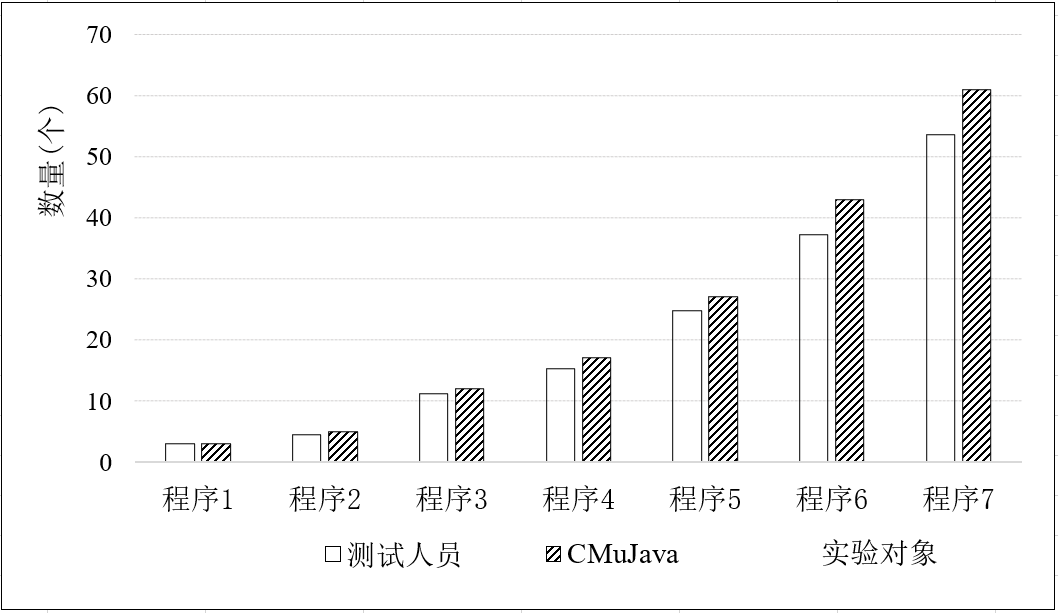


Fig. Number comparison chart of concurrent mutants

图 并发变异体的数量对比图

表4列出了5个测试人员针对不同待测程序生成并发变异体所使用的时间数据,从表4中可以看出随着程序复杂度的增加,测试人员所需要的时间也随之增加,而且即使针对同一个程序不同的测试人员所需要的时间也不完全相同.表5列出了CMuJava生成并发变异体所需要消耗的时间,由于工具所需要消耗的时间与计算机系统的资源分配有一定的关系,为了让实验数据更加合理,针对每一个待测并发程序都进行了5次实验并记录了每一次实验的用时数据.从表中可以看出,随着待测程序的复杂度增加工具所需要的时间也逐渐增加,而且每一次所需要的时间之间也有一定的差距.

**Table 4** Time for testers to generate concurrent mutants(second)

表4测试人员生成并发变异体用时(秒)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试者 | 程序1 | 程序2 | 程序3 | 程序4 | 程序5 | 程序6 | 程序7 |
| tester1 | 512 | 555 | 1004 | 1638 | 2903 | 4553 | 6765 |
| tester2 | 523 | 545 | 944 | 1701 | 3251 | 4452 | 7219 |
| tester3 | 448 | 508 | 900 | 1534 | 2953 | 4353 | 6891 |
| tester4 | 487 | 540 | 893 | 1758 | 3361 | 4616 | 6881 |
| tester5 | 524 | 582 | 987 | 1880 | 3097 | 4520 | 6098 |

**Table 5** Time for CMujava to generate concurrent mutants(second)

表5 CMujava生成并发变异体用时(秒)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 次数 | 程序1 | 程序2 | 程序3 | 程序4 | 程序5 | 程序6 | 程序7 |
| 1 | 0.063 | 0.077 | 0.14 | 0.25 | 0.581 | 0.984 | 1.187 |
| 2 | 0.07 | 0.077 | 0.156 | 0.234 | 0.503 | 0.921 | 1.321 |
| 3 | 0.054 | 0.079 | 0.098 | 0.25 | 0.519 | 0.962 | 1.191 |
| 4 | 0.052 | 0.088 | 0.12 | 0.232 | 0.499 | 0.906 | 1.201 |
| 5 | 0.062 | 0.09 | 0.109 | 0.249 | 0.523 | 0.895 | 1.296 |

图 给出了测试人员和CMuJava生成并发变异体用时对比盒图,横轴表示两种方式下的7个实验对象,纵轴表示所用时间,框的上下限表示所用的最多时间和最少时间.从图中可以明显地看出测试人员和CmuJ-

ava所需要消耗的时间形成了鲜明的对比.针对所有的程序,人工生成并发变异体所需要的时间都是CmuJa-

va的上千倍,而且随着程序的复杂度和可生成的并发变异体数量增加,测试人员和CMuJava的耗时差距越来越大.另一方面,人工生成变异体需要的时间也不稳定,和测试人员的个人能力有着很强的关系.使用CmuJava可以极大地提高生成并发变异体效率,降低并发变异测试的测试资源开销.

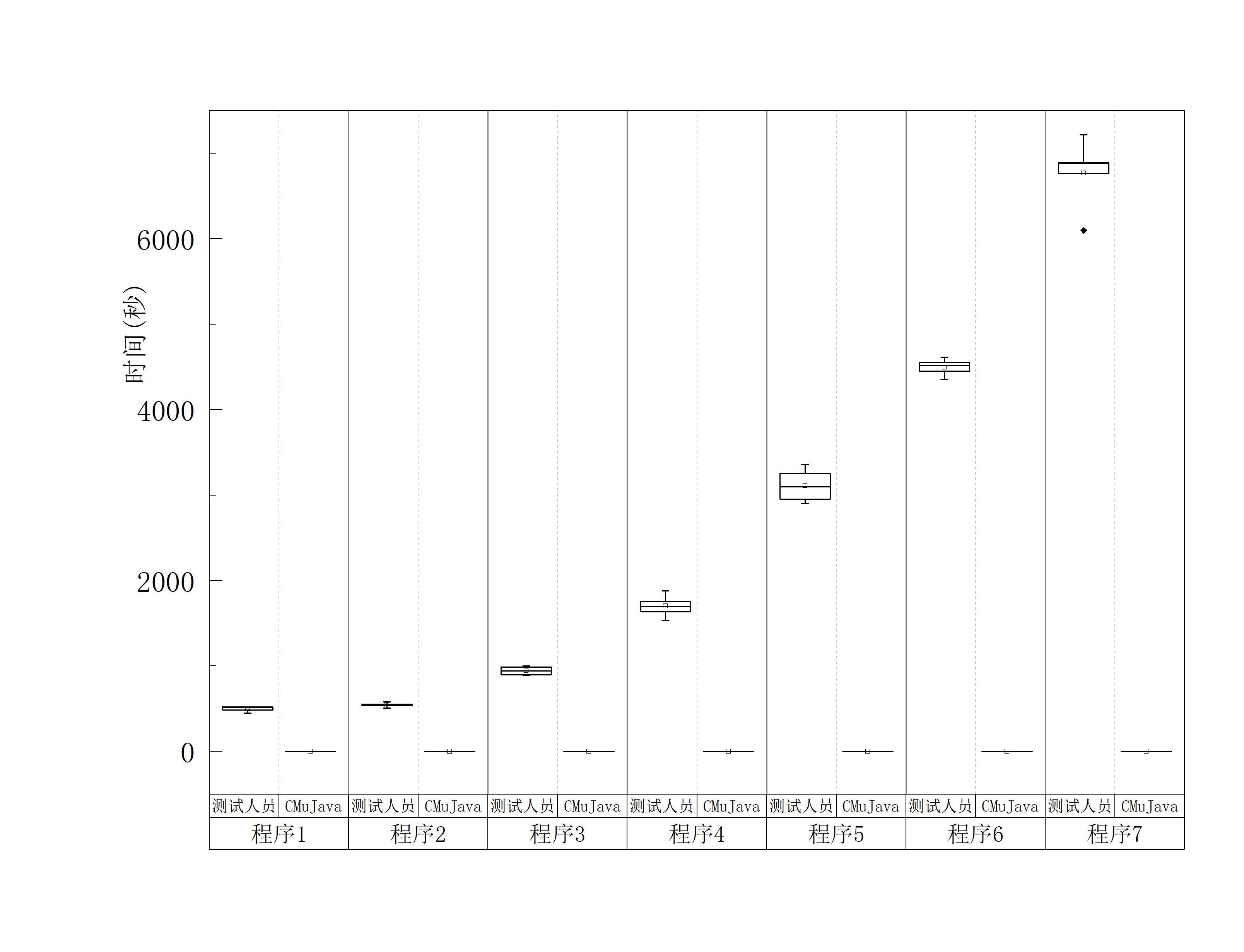


Fig. Boxplots of time spent on generating concurrent mutants

图 生成并发变异体所用时间盒图

## 分析

通过使用几个实际的并发程序对人工和工具生成并发变异体两种方式进行经验研究,研究结果表明,使用CMuJava可以显著地提升面向Java应用程序的并发变异测试自动化程度,降低测试资源的开销和提升生成并发变异体的效率.

在生产并发变异体的正确率方面,经验研究表明,相比于人工生成方式使用CMuJava可以将平均正确率提升约4%.对于一些容易出错的并发变异体(例如RCXC和ASK)提升的比例会更高;在生成并发变异体的完备性方面,使用CMuJava可以有效地避免人工生成并发变异体方式遗漏的数量也会随之逐渐增加的问题,保证能够准确识别待测程序的所有可变异点并生成相应的并发变异体,经验研究表明,相比于人工生成方式,使用CMuJava可以将生成并发变异体的数量提升约10%;在生成并发变异体效率方面,通过经验研究结果可以看出,相比于人工生成并发变异体的方式,使用CMuJava可以将生成效率平均提升约5700倍.CMuJava极大地提升了并发变异体生成的自动化程度,大幅提升了生成并发变异体的效率.

基于经验研究结果,相比于人工生成并发变异体,使用CMuJava可以显著地提高生成并发变异体的正确性、完备性和效率.

# 相关工作

从对并发变异算子的研究和变异测试工具的研究两方面介绍与本文相关的研究工作.

## 并发变异算子方面

在面向Java程序的并发变异算子研究方面,Delamaro等人根据Java语言的并发机制设计了15个并发变异算子[6],并且根据这些并发变异算子所针对的变异对象不同,将这些并发变异算子分成了针对监听对象、针对并发方法调用和针对等待集合三类.Farchi等人根据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[7].Brabury等人对Farchi等人提出的并发故障模式进行了补充,然后以这些并发故障模式为依据,设计出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等五类25个并发变异算子[8].Leon Wu和Gail Kaiser在分析了这些并发变异算子后,认为已有的并发变异算子无法生成某些并发变异体.他们对已有的变异算子进行组合得到了6个新的并发变异算子,并根据这些并发变异算子的特点分成了针对同步方法和针对同步代码块两类[9].

## 变异测试工具方面

变异算子是变异测试的一个核心概念,依据变异算子生成变异体在变异测试过程中至关重要.考虑到程序的复杂性,人工生成变异体会耗费大量的人力物力,测试人员需要一个自动生成变异体的工具来提高测试执行效率.

MuJava[10]是一个面向Java程序的变异测试工具,实现了方法级别和类级别的变异算子[17][18],但是并不支持并发变异算子的生成. Javalance为Java程序提供了开源的变异测试框架[12],但同样不支持并发变异算子的生成.本文提出了一种扩展了MuJava功能的工具,既能生成传统的方法和类级别的变异算子,又能生成25种并发变异算子[10].

……

# 总结

变异测试可以评估测试用例集的充分性与测试技术的有效性.并发变异测试是基于并发故障的变异测试技术.由于缺少面向Java程序的开源并发变异测试工具,研究者通常采用需要消耗大量测试资源的人工方式生成并发变异体.本文根据Bradbury等人提出的Java并发变异算子[8],设计并实现了一个面向Java程序的并发变异体自动生成工具CMuJava,提高了并发变异测试的自动化程度,具体来说:

1. 设计并实现了面向Java程序的并发变异体生成工具CMuJava:该工具扩展了传统的变异测试工具MuJava,支持待测程序选择、变异算子选择、并发变异体生成、并发变异体查看和测试报告打印等功能.
2. 对并发变异体生成工具CMuJava进行经验研究:利用7个并发程序评估并比较了CMuJava生成并发变异体和人工生成并发变异体的正确性、完备性和效率.实验结果表明:使用CMuJava可以显著提高生成并发变异体的正确性、完备性和效率.

我们将在以下几个方面进一步完善我们在面向Java的并发变异体生成工具的研究:(1)采用更多程序进行经验研究.本文使用了七个真实的并发程序进行经验研究,研究对象数量较少,未来需要使用更多的实验对象评估本文所开发的工具的实用性.(2)完善CMuJava的功能和性能.CMuJava目前还只是一个用于并发变异测试研究的原型工具,存在许多的不足之处,例如界面设计不够友好、运行性能较低、不支持执行测试用例和计算变异评分等问题.未来需要在此原型工具的基础上进行改进,完善系统的功能和提升性能,让CMuJava更好地运用到并发变异测试研究领域,提高面向Java程序的并发变异测试技术的自动化程度.

References:

1. Jia Y, Mark H. An analysis and survey of the development of mutation testing. IEEE Trans. on Software Engineering. IEEE Computer Society, 2011, 37(5): 649-678.
2. Myers GJ, Sandler C, Badgett T. The art of software testing. New Jersey: John Wiley & Sons, 2011.
3. Demillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. Computer, 1978, 11(4):34-41.
4. Carver R. Mutation-based testing of concurrent programs //Proceedings of International Test Conference (ITC 1993). IEEE Computer Society, 1993: 845-853.
5. Wu YB, Guo JX, Li Z, Zhao RL.Mutation strategy based on concurrent program data racing fault. Joumal of Computer Application, 2016(11): 3170-3177, 3195(in Chinese with English abstract).
6. Delamaro M, Pezzè M, Vincenzi A, Maldonado JC. Mutant operators for testing concurrent java programs //Proceedings of the Brazilian Symposium on Software Engineering (SBES 2001). IEEE Computer Society, 2001: 272-285.
7. Farchi E, Nir Y, Ur S. Concurrent bug patterns and how to test them //Proceedings of the International Parallel and Distributed Processing Symposium (PDPS 2003). IEEE Computer Society, 2003: 286-292.
8. Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrent java (J2SE 5.0) //Proceedings of the 2th Workshop on Mutation Analysis, Co-located with the IEEE International Symposium on Software Reliability Engineering (ISSRE 2006). IEEE Computer Society, 2006: 83-92.
9. Leon W, Kaiser G. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators //Proceedings of the 23th International Conference on Software Engineering and Knowledge Engineering (SEKE 2011). DBLP, 2011: 244-249.
10. Ma YS, Offutt AJ, Kwon YR. MuJava: An automated class mutation system. Software Testing Verification and Reliability, 2005, 15(2): 97-133.
11. Smith B, Williams L. On guiding the augmentation of an automated test suite via mutation analysis. Empirical Software Engineering, 2009, 14(3): 341-369.
12. Schuler D, Zeller A. Javalanche: Efficient mutation testing for java //Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE 2009). ACM Press, 2009: 297-298.
13. Goodenough J B. Toward a Theory of Test Data Selection. IEEE Transactions on Software Engineering, 1975, 1(1): 156-173.
14. 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).
15. Fang TF, Wei P, Cheng XM. The art of java concurrent programming. Machinery Industry Press, 2015.
16. Tatsubori M, Chiba S, Killijian M O, Itano K. OpenJava: A Class-based Macro System for Java //Proceedings of the Workshop on Reflection and Software Engineering, Co-located with the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1999). Springer, 1999: 117-133.
17. Ma YS. Description of method level mutation operators for java. Electronics and Telecommunications Research Institute. Korea, Tech.Rep..2015.
18. Offutt J, Ma YS, Kwon YR. The class-level mutants of MuJava //Proceedings of International Workshop on Automation of Software Test. ACM press, 2006: 78-84.
19. Gramoli V. More than you ever wanted to know about synchronization synchrobench, measuring the impact of the synchronization on concurrent algorithms //Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015). ACM Press, 2015: 1-10.
20. <https://logging.apache.org/log4j/2.x/>.
21. Andrews GR, Sechneider FB. Conceots and notations for concurrent programming. Comput. Surv. 1983, 15(1):3-43.

附中文参考文献:

[5] 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.

[14] 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] 方腾飞, 魏鹏, 程晓明. Java并发编程的艺术. 机械工业出版社, 2015.