摘 要: 并发系统由多个并发执行的流程组成,并且流程之间通常显示或隐式地共享一些存储空间.执行流之间执行次序的不确定性给并发系统的测试带来了严峻的挑战.并发变异测试是一种基于并发故障的软件测试技术,依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,通过执行并发变异体来评估测试用例集的充分性和测试技术的有效性.然而,面向Java程序的开源变异测试工具不支持生成并发变异体,人工生成并发变异体需要消耗大量的测试资源.为了提高并发变异测试的效率,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,并采用经验研究评估了CMuJava生成并发变异体的正确性、完备性及效率.实验结果表明,该工具生成并发变异体的正确性和完备性均达到了100%,并且与人工生成并发变异体相比,生成并发变异体的效率提高了大约5700倍.

关键词: 并发系统;并发变异测试; 并发变异算子; 并发变异体; 测试工具

**Abstract**: Concurrent systems are composed of multiple concurrent execution flows, which usually share storage space explicitly or implicitly. Uncertainty of execution order among execution flows makes testing concurrent systems a challenging task. Concurrent mutation testing is a software testing technology based on concurrent faults, which changes the syntax of the program under test according to concurrent mutation operators, generates concurrent mutants to evaluate the adequacy of test case set and the effectiveness of test technology. However, current open source Java-oriented mutation testing tools do not support concurrent mutants generation, and manual generation of concurrent mutants requires a lot of test resources. In order to improve the efficiency of concurrent mutation testing, we designs and implements a concurrent mutation automatic generation tool CMuJava in this paper, which is based on Java concurrent mutation operators, and evaluates the correctness, completeness and efficiency of CMuJava generating concurrent mutants by empirical research. The results of the empirical study show that the correctness and completeness of the tool for generating concurrent mutants are 100%, and the efficiency of generating concurrent mutants is about 5700 times higher than that of generating concurrent mutants manually.

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

随着多核计算的日益普及,并发程序已经成为人们广泛关注与重视的研究领域.并发程序中存在多个并发执行的流程,流程之间通常显示或者隐式地共享一些存储空间,并且它们的执行次序不确定[1].并发流程之间不确定的相互作用与影响的情形称为执行交错.执行交错的存在使得在完全相同的环境下运行两次同样程序的输出结果可能不同,并且程序故障难以重现,给并发程序的测试带来了严峻的挑战.如何有效地检测并发程序中潜藏的故障,提高并发程序的可靠性成为一个亟待解决的重要问题.

变异测试[2]是一种基于故障的软件测试技术,可以用来评估测试用例集的充分性与测试技术的有效性.测试人员首先通过分析待测程序设计出一系列变异算子[3],然后对待测程序应用变异算子来生成大量的变异体[3],在识别出等价变异体[3]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来提高测试充分性.

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中,提出了并发变异测试[29].与传统顺序程序的变异测试技术不同,并发变异测试应用并发变异算子来改变待测程序的语法结构,生成并发变异体.并发变异算子作为并发变异测试的基础,定义了改变待测并发程序语法结构的规则.

目前并发变异测试的相关研究处于起步阶段,研究者将注意力主要集中在生成新的并发变异算子方面[9].在并发变异体生成方面,面向Java语言的开源变异测试工具(例如MuJava[4]、Javalance [5]和Jumble [6]等)不支持生成并发变异体,而已有的面向Java并发程序的变异体生成工具要么实现了少量并发变异算子,要么不支持开源使用.研究者在测试过程中通常人工生成并发变异体,开销巨大.即使使用工具,由于一些并发变异算子产生的变异体数量很大,导致测试过程缓慢效率低.另外,随着并发系统的不断发展,并发程序越来越复杂,可能需要新的并发变异算子来模拟并发故障,因此新的并发变异测试工具要能够极大提高变异体生成效率,同时具有一定的可扩展性.

Brabury等人[7]依据Java(J2SE 5.0)并发机制特性及Java并发程序的故障类型[32],提出了面向Java程序的25种并发变异算子,反应了真实的并发故障,具有代表性和良好的覆盖性.本文根据Brabury等人提出的25种并发变异算子,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等人提出的并发变异算子[7] 扩展了面向Java程序的变异测试工具MuJava,实现了一个在能够生成传统变异体的基础上自动生成并发变异体的工具CMujava.
2. 采用MSG(Mutant Schema Generation)技术提高了CMuJava生成并发变异体的效率.
3. 实现过程通过使用Visitor设计模式提高了CMuJava的可扩展性.
4. 以经验研究的方式评估了CMuJava生成并发变异体的正确性、完备性和效率.

本文第一部分介绍了本文研究工作的相关背景,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评估工具生成并发变异体的正确性、完备性和效率,第四部分列出了实验中主要的有效性威胁,实验结果与分析展示在第五部分,第六部分介绍与本文相关的研究工作,第七部分对本文的工作做出了总结并提出将来需要完善的方面.

# 研究背景

## 传统变异测试

图1描述了传统变异测试的基本流程,该框架可以用来定量评估测试用例集充分性和测试技术的有效性.首先,应用变异算子来改变原始程序的语法结构,生成与原始程序有语法差异的变异体集合来模拟待测程序可能存在的故障.然后,在原始程序和每个变异体上执行所有的测试用例,如果存在测试用例在和上的执行结果不同,则称变异体为可杀死变异体,如果所有的测试用例在和上执行的结果完全一致,则称为的等价变异体.最后,通过式(1)计算变异得分,评估测试用例集的充分性和测试技术的有效性.



Fig.1 Mutation Testing process

图1 变异测试流程

(1),

其中,MS(M,T)表示变异得分,killed(M,T)表示可杀死变异体数量,|M|表示生成的所有变异体数量,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或测试技术越有效.

## 并发变异测试

与顺序程序相比,并发程序中多个执行流之间不确定的执行交错为测试并发程序带来严峻的挑战.很多研究者从不同的角度出发,针对并发程序提出了一系列故障检测技术,例如:锁集分析[29]、happens-before[30]和并发变异测试技术[31].并发变异测试将传统变异测试技术应用到并发程序中,模拟真实的并发故障,可以有效地提高故障检测效率.

传统变异测试定义为5元组[26],其中,P是原始程序,S是规格说明,D是测试用例集,,表示P中语句的位置,,是位置处的变异算子集合.

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.在传统变异测试模型的基础上,我们用C表示P中的并发机制所在的位置,将并发变异测试定义为6元组,其中,P是原始并发程序,S是规格说明,D是测试用例集,,表示P中语句的位置,,表示P中并发机制的位置,,是位置的变异算子集合,是变异算子的数量.

图2描述了并发变异测试的流程,主要步骤如下:

1. 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来生成相应的并发变异体.
2. 从生成的大量并发变异体中识别出等价并发变异体并删除.
3. 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得分.如果变异得分满足要求则变异测试结束,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变异得分满足要求.



Fig.2 Concurrent Mutation Testing process

图2 并发变异测试流程

# CMuJava的设计与实现

为了提高并发变异测试的自动化程度,我们开发了面向Java并发程序的变异体生成工具CMuJava.该工具基于MuJava 提供的应用程序接口API以及并发变异算子的变异规则,在生成传统变异体的基础上,能够生成并发变异体.

## 功能

CMuJava并发变异体生成工具的主要功能如下:

1. **源程序分析.**分析待测程序语法和结构信息,输出元对象.
2. **变异体生成.**依据变异算子定义的规则改变语句语法信息生成变异体,记录每个变异体修改的信息,输出变异体和变异日志.
3. **变异体查看.**根据系统配置信息获取待测程序和变异体所在路径,展示选中的待测程序的所有变异体信息,包括被标记过的变异体和原始待测程序.
4. **变异体执行.**对变异体执行测试用例,输出变异得分以及每一个变异体的“杀死”信息.

## 系统框架

图3描述了CMuJava的基本框架.

1. **源程序分析.**该模块通过OpenJava的Compile-time reflection首先读取待测程序及其依赖程序，按照Java程序的语法结构拆分并进行语法分析,得到待测程序传统的语法信息和程序中的并发机制.将程序的变量信息封装到一个变量表里,便于统一管理并对外提供相应元素的操作API,最后将待测程序的所有语法信息和逻辑结构信息按照Java程序的组织结构封装到一个元对象中.该模块最后输出一些列元对象用于变异体生成.
2. **变异体生成.**该模块根据测试人员选择的变异算子动态拼接相应的变异算子类名,利用反射技术实例化相应的变异算子类对象,遍历元对象的节点,判断是否存在可以应用所选变异算子的语句;然后依据变异算子所定义的规则,对符合条件的语句的语法结构更改生成变异后的元对象.该模块采用了MSG(Mutant Schema Generation)技术得到每个变异算子对应的元变异体,对元变异体进行语法检查,将正确的元变异体实例化为变异体并输出.
3. **变异体执行.**该模块调用Java反射机制的API对生成的变异体执行测试人员编辑的JUnit测试脚本,自动获取测试结果,统计最终的变异得分以及每一个变异体的“杀死”信息.



Fig.3 Structure of CMujava

图3 CMujava架构

## 实现的关键技术

CMuJava通过采用MSG(Mutant Schema Generation)技术[8]生成元变异体(Matemutant)[8]的方法提高并发变异体生成效率.另外,为了提高工具的可扩展性,我们在实现过程中采用了Visitor设计模式.

(1)元变异体生成

变异分析方法需要对同一程序的所有变异体分别进行解释,一些变异算子应用后会生成大量变异体,导致分析速度缓慢.为了提高变异测试的执行效率,研究者提出了MSG方法.MSG技术可以将一个程序的所有变异体都编码到一个特殊的参数化程序中,这个特殊的参数化程序叫做元变异体.元变异体包含了待测程序的所有变异体信息,运行过程中,元变异体具有待测程序所有变异体的功能.

我们以AOR变异算子为例简要介绍MSG技术的原理和实现.AOR变异算子所定义的规则就是对待测程序内的算数运算符进行替换.例如,对于算数表达式C=A+B.可应用AOR变异算子生成如下几个变异体:

C=A-B;

C=A\*B;

C=A/B;

C=A%B.

这些变异体可以抽象为如式(2)所示的更通用的表达式:

C=A op B (2),

其中op为算数运算符的抽象表示.还可以再进一步将表达式(2)重写为一个符合一般编程语言语法规范的表达式,如式(3)所示:

C = fun\_aor(A,B,op) (3),

其中fun\_aor为一个具体的方法,可以实现5种可能的算数运算符中任意一种.其实现算法详见图3.该类方法包含了表达式的所有的变异体信息,只需要传入相应的操作符和操作数即可生成相应的变异体.将程序中的语句改变成这种形式后的参数化程序即为元变异体.

|  |
| --- |
| Algorithm 1 AOR |
| **Input:** A, B, operator  1. **switch** operator  2. case ADD  3. expression = A + B  4. break  5. case DEC  6. expression = A - B  7. break  8. case MUL  9. expression = A \* B  10. break  11. case DIV  12. expression = A / B  13. break  14. case MOD  15. expression = A % B  16. break  17. **end switch**  18. **return expression** |

Fig.3 Implementation algorithm of AOR metamutant

图3 AOR元变异体实现算法

MuJava应用了MSG方法来提高变异测试的执行效率[4].本文为了提高并发变异测试的执行效率,同样使用了MSG方法来进行变异分析.下面以并发变异算子MSP(Modify Synchronized Parameter)为例介绍一下MSG技术在CMuJava中的使用.MSP变异算子所定义的规则就是更改synchronized代码块的同步参数,将其替换为当前类内部定义的其他成员变量或者this关键字.图4给出了三个具体的MSP变异体的例子.

从MSP的变化规则可以知道,每次被更改的地方都只有synchronized的同步参数.将以上源程序的synchronized代码块进行抽象可以得到如式(4)所示的表达式:

synchronized(getParameter(orignalArg, objList)){…} (4),

这样在程序运行的时候就可以通过getParameter方法动态的获取同步参数并生成相应的变异体. getParameter方法有两个参数,其中originalArg表示的是原来的同步参数,objList表示的是可以用于替换原来的同步参数的对象列表.getParameter方法的实现核心算法如图5.

|  |  |
| --- | --- |
| 源程序：  ...  private Object obj1 = new Object();  private Object obj2 = new Object();  private Object obj3 = new Object();  ...  synchronized(this){...}  ... | MSP变异体1：  ...  private Object obj1 = new Object();  private Object obj2 = new Object();  private Object obj3 = new Object();  ...  synchronized(obj1){...}  ... |
| MSP变异体2：  ...  private Object obj1 = new Object();  private Object obj2 = new Object();  private Object obj3 = new Object();  ...  synchronized(obj2){...}  … | MSP变异体3：  ...  private Object obj1 = new Object();  private Object obj2 = new Object();  private Object obj3 = new Object();  ...  synchronized(obj3){...}  … |

Fig.4 Example of MSP mutant

图4 MSP变异体示例

|  |
| --- |
| Algorithm 2 MSP |
| **Input:** originalArg, objList  1. **for** obj in objList  2. **if** obj ≠ orignalArg  3. generate MSP mutant with obj.  4. **end if**  5. **end for** |

Fig.5 Implementation algorithm of MSP metamutant

图5 MSP元变异体实现算法

在对一个并发程序应用MSP并发变异算子时,CMuJava在确定当前程序符合MSP并发变异规则之后,会自动获取程序的所有成员变量.然后将synchronized代码块的原始参数和成员变量列表传递给此方法,就可以生成待测程序的所有MSP变异体.

(2)设计模式应用

并发变异算子可以模拟程序中真实的并发故障,随着并发程序的发展,可能会发现新的并发故障,因此会出现新的并发变异算子.我们提出的CMuJava工具目前实现了25种并发变异算子,CMuJava在完成一个Java程序解析得到一个元对象之后,需要在此对象的各个节点上运用一系列并发变异算子进行可变异点查找,然后对可变异点进行语法更改生成并发变异体.如果将所有的可变异点判断逻辑和变异体输出逻辑都放到元对象的每个节点内部,每增加一个变异算子都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以完成此变异算子的功能,这样会导致相关节点的代码变得越来越复杂,在降低了代码的可读性的同时也会增加后期的维护成本,同时不易于扩展.

Visitor设计模式[10]适用于被访问对象结构相对稳定,且访问者操作不确定的情况.CMuJava系统所涉及的元对象的数据结构非常稳定,不存在需要新加入节点或者对已有节点进行修改的情况,但是存在到对元对象的大量不确定操作,涉及到众多并发变异算子.为了提高工具的可扩展性,便于添加新的变异算子,我们在CMuJava实现过程中采用了Visitor设计模式.

图6描述了CMuJava的核心类结构.其中元对象(MetaObject)内各个节点作为被访问对象,并发变异算子作为具体的访问者(visitor).当有新的并发变异算子出现需要添加到系统中时,只需要编写新的访问者(visitor)类来访问数据结构即可,极大地提高了工具的可扩展性.



Fig.6 Implementation algorithm of MSP metamutant

图6 MSP元变异体实现算法

## CMuJava实现的并发变异算子

目前CMuJava支持的25种并发变异算子如下[[1]](#footnote-1):

(1) MXT(Modify Method-X Timeout).修改并发机制wait(),sleep(),join()方法的超时参数,将其变为原来的一半或者两倍.例如,wait(t)替换为wait(t/2)或者wait(t\*2).

(2) MSP(Modify Synchronized Block Parameter).修改synchronized代码块参数,将其由this关键字替换为其他对象,或者由其他对象修改为this关键字.例如,synchronized(this)替换为synchronized(lock).

(3) ESP(Exchange Synchronized Block Parameters).如果代码中存在两个synchronized代码块嵌套情况,则交换这两个synchronized代码块的参数.例如,synchronized(lock1){synchronized(lock2){…}}替换为synchronized(lock2){synchronized(lock1){…}}

(4) MSF(Modify Semaphore Fairness).修改信号量公平参数.例如,Semaphore(permits,true)替换为Semaphore(permits,false).

(5) MXC(Modify Concurrency Mechanism-X Count).修改Semaphore、Latch和Barrier的初始化数量,将其初始化数量增加1或者减少1.例如,p替换为p- -或者p+ +.

(6) MBR(Modify Barrier Runnable Parameter).删除CyclicBarrier()方法的可选参数.

(7) RTXC(Remove Thread Method-X Call).删除wait(),join(),sleep(),yield(),notify(),notifyAll()等方法调用.

(8) RCXC(Remove Concurrency Mechanism Method-X Call).删除以下并发机制的指定方法调用,Locks (lock(),unlock()),Condition(signal(),signalAll()),Semaphore(acquire(),release()),Latch(countDown())和ExecutorService(如submit()).

(9) RNA(Replace NotifyAll() with Notfiy()).用notify()方法替换notifyAll()方法.

(10) RJS(Replace Jon() with Sleep()).用sleep()方法替换join()方法.

(11) ELPA(Exchange Lock/Permit Acquistion).交换获取锁或者权限的方法.例如,sem.acquire()替换为sem.acquireUninterruptibly()或者sem.tryAcquire().

(12) SAN(Switch Atomic Call with Non-Atomic).将原子变量的getAndSet()方法调用拆分为get()和set()两个方法进行调用.

(13) ASTK(Add Static Keyword to Method).向方法添加static关键字.例如,public synchronized void aMethod(){…}修改为public static synchronized void aMethod(){…}.

(14) RSTK(Remove Static Keyword from Method).删除static关键字.例如,public static synchronized void bMethod(){…}修改为public synchronized void bMethod(){…}.

(15) ASK(Add Synchronized Keyword from Method).添加synchronized关键字.例如,public void cMethod(){…}修改为public synchronized void cMethod(){…}.

(16) RSK(Remove Synchronized Keyword from Method).删除方法定义里的synchronized关键字.例如,public void synchronized dMethod(){…}修改为public void dMethod(){…}.

(17) RSB(Remove Synchronized Block).将synchronized代码块内的语句提取出来,使其不再被保护.

(18) RVK(Remove Volatile Keyword).删除变量定义时的volatile关键字.例如,volatile long x替换为long x.

(19) RFU(Remove Finally Around Unlock).如果finally代码块内存在unlock()方法,则将finally代码块内的语句提取出来,并删除finally代码块.

(20) RXO(Replace One Concurrency Mechanism-X with Another).当存在两个使用相同并发机制的实例化对象时,将这两个对象进行交换.

(21) EELO(Exchange Explicit Lock Object).交换两个显示锁对象.

(22) SHCR(Shift Critical Region).移动临界区.即向上或向下移动临界区代码块.

(23) EXCR(Expand Critical Region).扩展临界区.即将临界区的语句之上或之下的语句包括到临界区.

(24) SKCR(Shrink Critical Region).收缩临界区.即将临界区内部需要同步的语句移动到临界区以外.

(25) SPCR(Split Critical Region).拆分临界区.即将一个临界区拆分为两个独立的临界区.

## 工具演示

本文基于Java开发了工具原型,图8展示了工具各个模块的界面.在图8(a)中,界面左侧列出了所有待测程序,首先测试人员选择需要生成变异体的待测程序,图中以实验对象Bin程序为例.然后选择界面右侧显示的变异算子,其中最左侧为19个方法级别变异算子,中间为28个类级别变异算子,这两类变异算子功能由MuJava提供,最右侧列出了Brabury等人提出的25个并发级别变异算子.最后,测试人员需要点击最下方黄色的“Generate”按钮然后等待系统完成变异体生成工作,系统会依次读取选择的待测程序,为每个待测程序生成相应的元对象,然后在每个元对象上依次应用选中的变异算子生成相应的变异体并输出到指定目录.

变异体生成完成后,CMuJava提供了查看变异体的功能,查看方法变异体、类变异体和并发变异体的界面分别如图8(b),(c),(d)所示.另外CMuJava还集成了MuJava执行变异体的功能,通过选择测试用例执行生成的变异体,最终得到测试结果.

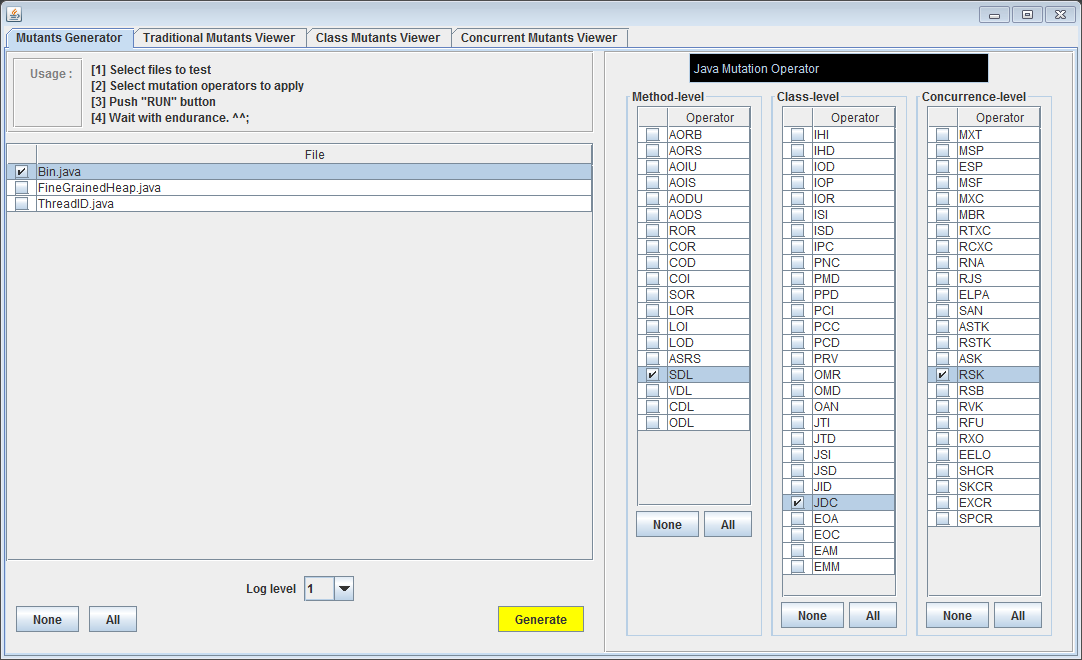
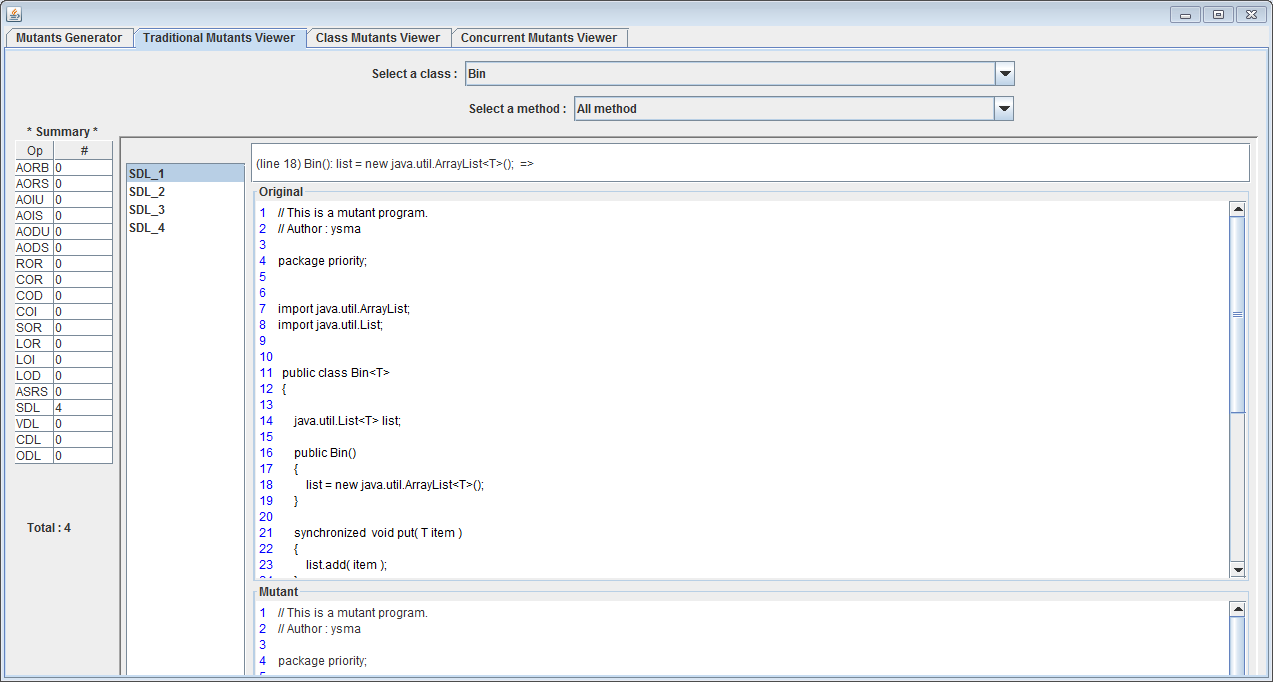
# 经验研究

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,本节对实验的设置和细节进行介绍.

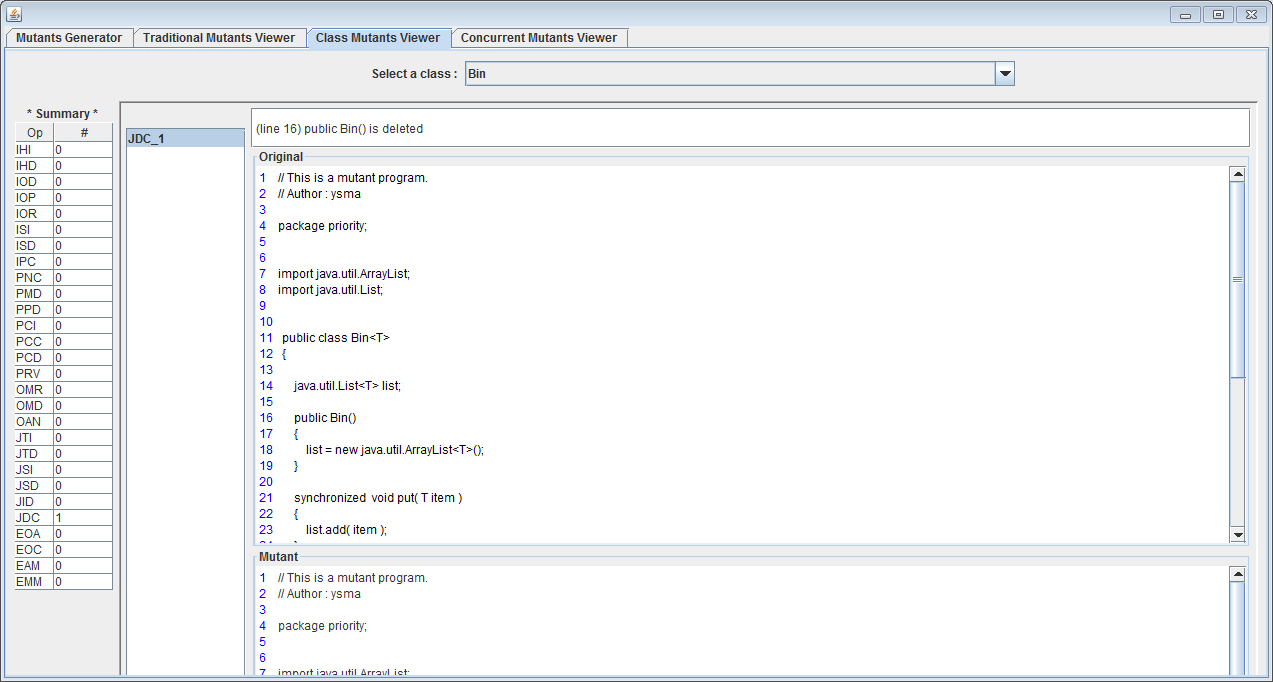
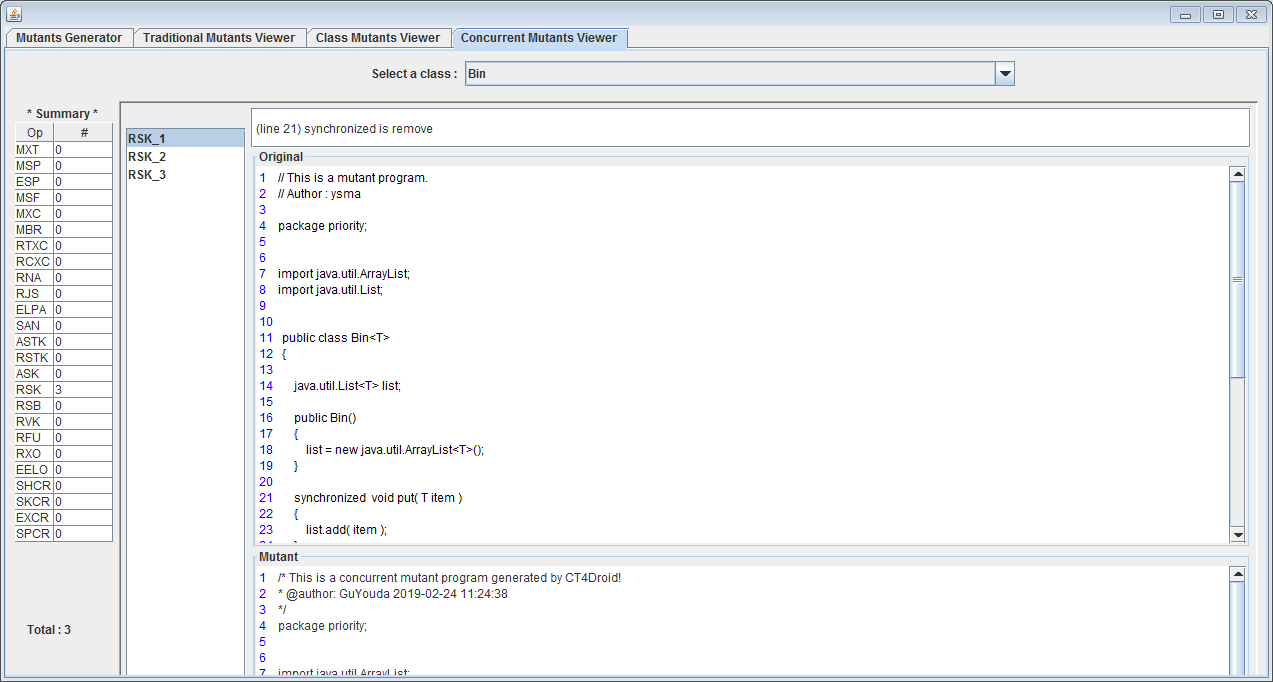
## 研究问题

通过实例研究,我们希望回答如下三个问题.

1. 使用CMuJava工具是否能够提高生成并发变异体的正确率?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有的并发机制并生成正确的并发变异体,然后计算正确率,将实验结果和人工生成的并发变异体的正确率进行对比分析,评估工具是否提高了正确率.
2. 使用CMuJava工具生成并发变异体是否能够提高生成的并发变异体的完备性?首先分析并发程序内部所使用的并发机制确定程序理论上应该生成的并发变异体的数量,将工具生成的变异体和人工生成的变异体数量分别与理论数量进行对比分析,评估工具生成变异体是否提高了完备性.
3. 使用CMuJava工具是否能够提高生成并发变异体的效率?记录工具生成所有并发变异体所需要的时间,然后将实验结果和人工生成并发变异体需要的时间进行对比分析,评估工具的运行效率.

(a) CMujava主界面 (b) 方法变异体查看界面

(c) 类变异体查看界面 (d) 并发变异体查看界面

Fig.8 GUI of CMuJava

图8 CMuJava界面

## 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7个并发程序:Bin[12]、ThreadID(TID) [12]、FineGrainedHeap(FGH) [12]、StripedSizedEpoch(SSE)、Logi-calOrderingAVL(LOAVL)、TranscationalFriendlyTreeSet(TFTS)和TmsManager(JM).7个并发测试基准程序涵盖了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.表2列出了每个实验对象的名字、代码行数以及可应用的并发变异算子.程序1~3是三个并发测试基准程序.程序4~6来自于一个专门的并发测试基准程序库Synchrobench[13],程序7来自于Apache开源库下的日志组件Log4j[[2]](#footnote-2).

**Table 2** The information of experimental object

表2 实验对象信息

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 程序名 | 代码行数 | 可应用的并发变异算子 |
| 1 | Bin | 42 | RSK |
| 2 | 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 |

## 变量

1. 自变量

针对前文提出的研究问题,本次经验研究的自变量为提出的并发变异体生成工具CMuJava生成变异体和测试人员人工生成变异体.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.

1. 因变量

本文采用正确率指标来评估生成并发变异体的正确性,其计算公式如式5所示,

(5).

采用统计生成并发变异体的数量的方法来评估生成并发变异体的完备性,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并进行比较.

通过记录CMuJava解析待测程序和生成并发变异体的时间,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,来评估CMuJava生成并发变异体的效率.

## 实验设置

1. 实验环境

CMuJava运行在64位Windows 10操作系统,主要配置为:Intel Core i7-7700HQ CPU、16GB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.

1. 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,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.

# 有效性威胁

经验研究采用的实验对象的正确性对实验结果有关键的影响.对于实验对象中Bin、ThreadID和FineGrainedHeap三个程序,课题组已经应用其做了大量相关实验,能够保证程序的正确性和代表性.

为了使实验更具说服力我们采用了对比试验的方法.对于研究过程中统计数据的正确性,实验中采用多次实验并求平均值的方法加以保证.

# 实验结果与分析

表3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4总结了两种方式生成并发变异体的数量.

**Table 3** Correct rate of concurrent mutants

表3 生成并发变异体正确率

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 生成变异体的对象 | Bin | TID | FGH | SSE | TFT | LOAVL | JM |
| 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% |

从表3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,部分测试人员会生成了一些语法错误的变异体.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)变异算子定义的变异体,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而导致无法运行的情况,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%.

图9是测试人员和CMuJava生成变异体的正确率对比图.横轴表示两种变异体生成方式下的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(如ThreadID),正确率的提高程度更为显著.

从表4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10是测试人员和CMuJava生成并发变异体的数量对比图.

**Table 4** Number of concurrent mutants

表4 生成并发变异体数量(个)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 生成变异体的对象 | Bin | TID | FGH | SSE | TFT | LOAVL | JM |
| 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,主要就是依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CMuJava在生成变异体的过程中保持稳定不会出现遗漏变异体的问题.



Fig.9 Correct rate comparison chart of concurrent mutants

图9 并发变异体正确率对比图



Fig.10 Number comparison chart of concurrent mutants

图10 并发变异体的数量对比图

表5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5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6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次实验并记录每一次实验的用时.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 5** Time for testers to generate concurrent mutants(second)

表5测试人员生成并发变异体用时(秒)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试者 | Bin | TID | FGH | SSE | TFT | LOAVL | JM |
| 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 6** Time for CMuJava to generate concurrent mutants(second)

表6 CMuJava生成并发变异体用时(秒)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 次数 | Bin | TID | FGH | SSE | TFT | LOAVL | JM |
| 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 |



Fig.11 Boxplots of time spent on generating concurrent mutants

图11 生成并发变异体所用时间盒图

从表中可以明显的看出CMuJava生成并发变异体的效率更高,针对所有的程序,人工生成并发变异体所需要的时间都是CMuJava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.为了清楚地看出每次实验结果的波动情况,我们应用公式(6)对两种方式生成并发变异体的用时数据做了标准化处理,其中x是生成每个实验对象的所有并发变异体的时间值,min和max分别对应最小值和最大值,表示标准化后的值.图11为处理后的数据对比盒图,横轴表示两种变异体生成方式下的7个实验对象,纵轴表示实验所用时间标准化后的值.从图中可以看出CMuJava生成并发变异体需要的时间相比人工生成的方式更加稳定.因此使用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.

(6).

通过使用几个实际的并发程序对人工和CMuJava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高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变异体的问题,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

自变异测试的方法被提出以来,测试人员在该领域做了大量的研究工作,涉及C/C++、C#、Java、SQL等多种编程语言[14].其中一些方面的工作为针对不同编程语言的变异算子的定义[7,11,15,16,17,18]、针对不同编程语言的变异体自动化生成工具[4,20,25],变异算子的选择,以及变异测试技术优化.本节介绍与本文提出的面向Java程序的并发变异体生成工具相关的工作.

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[15],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[16].Brabury等人[7]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共25个并发变异算子.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[17].Milos Gligoric等人在实验过程中提出了3种新的并发变异算子[21],并结合Brabury等人提出的25种并发变异算子一起应用到实验中.

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.随着变异算子的发展,大量针对不同编程语言的自动生成变异体的工具被实现,Papadakis等人对已有的变异测试工具做出了总结[14].其中包括面向Java语言的变异体生成工具[4,5,6].MuJava[4]是一个面向Java程序的变异测试工具,实现了传统的方法级别和类级别的变异算子[4,11],但是并不支持并发变异体的生成.Javalance[5]为Java程序提供了开源的变异测试框架,但同样不支持并发变异体的生成.Jumble[6]是一个变异Java程序字节码的工具,仅提供传统的方法级别的变异.目前已有的面向并发程序的变异工具[22,23,24]仍存在一些问题.MutMut[23]是一个面向并发程序的优化变异体执行的工具,据报道该工具对变异测试的时间降幅达77%,但MutMut并不具有优化变异体生成的功能.Paraµ[24]是一个支持并发变异体生成的工具,但是该工具的研究仍处于初级阶段,仅实现了2个类级别变异算子和3个并发变异算子.另一个面向Java程序的支持并发变异体生成的工具Comutation[21]实现了所有的28种并发变异算子,但是由于Intel没有对该工具授权发布,Comutation目前并不支持开源使用.

# 总结

变异测试是一种可以评估测试用例集的充分性与测试技术的有效性的技术.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程序的开源的并发变异测试工具,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[7],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,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、变异体执行和测试报告打印等功能,提高了并发变异测试的自动化程度.另外利用7个并发程序评估了CMuJava生成并发变异体的性能.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.

未来,我们将在如下几个方面进一步完善我们在面向Java程序的并发变异体生成工具方面的研究:(1)本文使用了7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提出的支持工具的实用性.(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,仍存在不足之处,例如界面设计不够友好.未来需要在此原型工具的基础上进行改进,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. 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. IEEE 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. Ma YS, Offutt AJ, Kwon YR. MuJava: An automated class mutation system. Software Testing Verification and Reliability, 2005, 15(2): 97-133.
5. Schuler D, Zeller A. Javalanche: Efficient mutation testing for java. In: Proc. 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.
6. 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 Testing: Academic and Industrial Conference Practice and Research Techniques – MUTATION. IEEE Computer Society. 2007: 169-175.
7. Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrent java (J2SE 5.0). In: Proc. 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.
8. Untch RH, Offutt AJ, Harrold MJ. Mutation analysis using mutant schemata. In: Proc. of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1993). ACM Press, 1993: 139-148.
9. 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).
10. Pati T, Hill JH. A survey report of enhancements to the visitor software design pattern. Software: Practice and Experience, 2014, 44(6): 699-733.
11. Offutt J, Ma YS, Kwon YR. The class-level mutants of MuJava. In: Proc. of International Workshop on Automation of Software Test. ACM press, 2006: 78-84.
12. Herlihy M, Shavit N. The art of multiprocessor programming. Morgan Kaufmann, 2012.
13. 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 Symposium on Principles and Practice of Parallel Programming. ACM Press, 2015: 1-10.
14. Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M. Mutation testing advances: An analysis and survey. Advances In Computer, 2019: 275-378.
15. Delamaro M, Pezzè M, Vincenzi A, Maldonado JC. Mutant operators for testing concurrent java programs. In: Proc. of the Brazilian Symposium on Software Engineering (SBES 2001). IEEE Computer Society, 2001: 272-285.
16. Farchi E, Nir Y, Ur S. Concurrent bug patterns and how to test them. In: Proc. of the International Parallel and Distributed Processing Symposium (PDPS 2003). IEEE Computer Society, 2003: 286-292.
17. Leon W, Kaiser G. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. In: Proc. of the 23th International Conference on Software Engineering and Knowledge Engineering (SEKE 2011). DBLP, 2011: 244-249.
18. Ma YS, Kwon YR, Offutt J. Inter-class mutation operators for java. In: Proc. of International Symposium on Software Reliability Engineering. IEEE, 2002: 352-366.
19. Smith B, Williams L. On guiding the augmentation of an automated test suite via mutation analysis. Empirical Software Engineering, 2009, 14(3): 341-369.
20. 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.
21. Gligoric M, Zhang L, Pereira C, Pokam G. Selective mutation testing for concurrent code. In: Proc. of the 2013 International Symposium on Software Testing and Analysis. ACM Press, 2013: 224-234.
22. Bradbury JS, Cordy JR, Dingel J. Comparative asscessment of testing and model checking using program mutation. In: Proc. of Testing: Academic & Industrial Conference Practice & Research Techniques-mutation. IEEE Computer Society, 2007: 210-219.
23. Cligoric M, Jagannath V, Marinov D. MutMut: Efficient exploration for mutation testing of multithreaded code. In: Proc. of the third International Conference on Software Testing. IEEE Press, 2010: 55-64.
24. Madiraju P, Namin AS. Paraµ: A partial and higher-order nutation tool with vonvurrency operators. In: Proc. of IEEE fourth Internation Conference on Software Testing. IEEE Press, 2011: 351-356.
25. Kusao M, Wanng C. Ccmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In: Proc. of the 28th IEEE/ACM International Conference on Automated Softwafre Engineering. 2013: 722-725.
26. Morell LJ. A theory of fault based testing. IEEE Trans. on Software Engineering. 1990, 16(8): 844-857.
27. Sun CA, Wang G, Cai KY, Chen TY. Distribution-Aware mutation analysis. In: Proc. of IEEE Computer Software & Applications Conference Workshops. IEEE Computer Society, 2012: 170-175.
28. Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrent java (J2SE 5.0). Technical report 2006-520, Queen’s University, 2006.
29. Lipton RJ. Reduction: A method of proving properties of parallel programs. Communications of the ACM, 1975,18(12): 717–721.
30. Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558-565.
31. Carver R. Mutation-based testing of concurrent programs. In: Proc. of International Test Conference(ITC 1993). IEEE Computer Society, 1993: 845-853.
32. Farchi E, Nir Y, Ur S. Concurrent bug patterns and how to test them. In: Proc. of 17th International Symposium on Parallel & Distributed Processing(IPDPS 2003). IEEE Computer Society, 2003: 286.2.

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.

[9] 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.

1. 本节对部分并发变异算子给出了简单的例子,可以阅读Brabury等人的技术报告. [↑](#footnote-ref-1)
2. https://logging.apache.org/log4j/2.x/ [↑](#footnote-ref-2)