摘　要: 并发系统由多个同时运行的执行流组成,系统中多个执行流之间存在的执行交错往往会导致程序中出现故障,执行交错的不确定性使得测试并发系统成为一项具有挑战性的工作.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.然而,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,并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 systems are composed of multiple execution flows that run simultaneously. The interleaving between multiple execution flows in the system often leads to program faults. The uncertainty of interleaving 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 the concurrent mutation operators. This technology can 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 this paper, we have designed and implemented a tool for generating conccurrent mutants automatically, CMuJava, according to mutation operators for concurrent Java, and conducted an empirical study to evaluate the correctness, completeness and efficiency of generating concurrent mutants with CMuJava.

Key words: 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产生大量的变异体[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.

并发变异测试是将变异测试理论和技术应用到并发程序中,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极其重要的作用.目前对并发变异测试的相关研究正处于起步阶段,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].另外,并发变异测试过程涉及到很多程序的执行,为了提高变异测试的执行效率,研究者提出了自动化的变异测试工具.然而目前已有的开源的面向 Java 语言的变异测试工具(例如 MuJava[4]、Javalance [5]和 Jumble [6]等)不支持生成并发变异体,已有的面向并发程序的变异体生成工具只能实现部分并发变异算子或者不支持开源使用,研究人员在测试过程中只能耗费大量测试资源来人工生成并发变异体.因此设计并实现一个能够实现自动生成并发变异体的工具成为一个亟待解决的问题.

本文根据 Brabury 等人提出的并发变异算子[7],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]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 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验证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作出了总结并提出将来需要完善的方面.

1 研究背景

本节简单介绍跟本文相关的背景知识.首先对变异测试的基本原理进行简单的介绍,然后介绍了并发变异测试相关信息.

1.1 变异测试

变异测试[2]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[3]来改变待测程序的语法结构,进而生成变异体[3].

（2） 从生成的大量变异体中识别出等价变异体[3].

（3） 在剩余的非等价变异体[3]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.

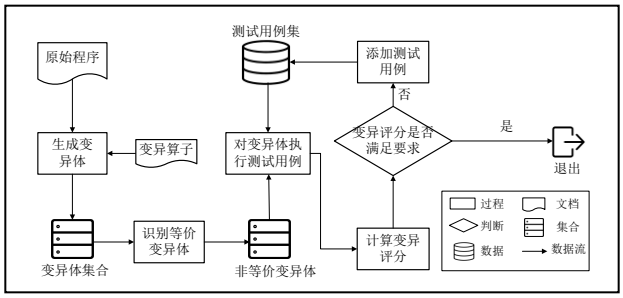


Fig.1 Mutation Testing process

图 1 变异测试流程

下面对上述流程中所涉及到的一些概念进行解释.

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

变异算子[3]: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 而生成了相应的变异体.

可以“杀死”的变异体[3]: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[3]: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[3]: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[3]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:

$$MS(M,T)=\frac{killed(M,T)}{\{|M|-eqv(M)\}} \quad (1),$$

其中,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[3].

1.2 并发变异测试

并发程序中存在多个并发执行的流程,流程之间通常显式或隐式地共享一些存储空间,它们之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,还需要流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[7]提出的面向 Java 程序的 25 种并发变异算子能够反应真实的并发故障,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) [7],我们将在本文第 2 部分详细介绍.

2 面向 Java 程序的并发变异体生成工具-CMuJava

本节首先列出 Brabury 等人提出的 25 种并发变异算子[7]的规则定义,然后介绍本文提出的支持工具的关键技术,最后展示本文提出的面向 Java 程序的并发变异体生成工具.

2.1 并发变异算子简介

Brabury 等人提出的 25 种并发变异算子[7]详见表 4.25 种并发变异算子共分为 5 类.

Table 1 concurrent mutation operators for java

表 1 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.具体包括六种变异算子:

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,交换相邻的两个锁对象.

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.

2. 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.具体包括六种变异算子:

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(),Condition(),Semaphore(),Latch()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()及交换锁获取方法 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.

3. 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:ASK、RSK、ASTK、RSTK、RSB、RVK 和 RFU 六种变异算子.

4. 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.

5. 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.具体包括四种变异算子:

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引起故障.

2.2 MSG方法

传统的变异测试分析方法需要对同一程序的许多细微变化分别进行解释,导致分析速度缓慢.为了提高变异测试的执行效率,研究人员提出了 MSG(Mutant Schema Generation)方法[8].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[8].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:

$$C=A\ op\ B \qquad (2),$$

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:

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

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.其实现算法伪代码详见图 4.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 2 AOR
Input: A, B, operator
Output: expression
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.4 Implementation algorithm of AOR metamutant

图 4 AOR 元变异体实现算法

已有的变异工具 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 5 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 6.

源程序：	MSP 变异体 1：

...
private Object obj1 = new Object();
private Object obj2 = new Object();
private Object obj3 = new Object();
...
synchronized(this){...}
...
	...
	private Object obj1 = new Object();
	private Object obj2 = new Object();
	private Object obj3 = new Object();
	...
	synchronized(obj1){...}
	...

MSP 变异体 2：	**MSP 变异体 3：**

...
private Object obj1 = new Object();
private Object obj2 = new Object();
private Object obj3 = new Object();
...
synchronized(obj2){...}
…
	...
	private Object obj1 = new Object();
	private Object obj2 = new Object();
	private Object obj3 = new Object();
	...
	synchronized(obj3){...}
	…

Fig.5 Example of MSP mutant

图 5 MSP 变异体示例

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.

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

Fig.6 Implementation algorithm of MSP metamutant

图 6 MSP 元变异体实现算法

2.3 Visitor设计模式

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.

由于 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,但是需要涉及到对元对象的大量不确定操作.为了增加系统代码的可读性和系统的可维护性,本文使用了 Visitor 设计模式[10]进行实现,很好地将系统数据结构和在相应数据结构之上的操作进行分离.

Visitor 模式需要涉及到抽象节点、具体节点、抽象访问者、具体访问者、结构对象和客户端五类角色,这些不同的角色相互作用才能完成系统的预期功能.

抽象节点(Abstract Node):所有具体节点的父类,声明了一个或多个 accept 方法,并使用一个访问者对象作为参数,通常使用抽象类或者接口的方式实现.本系统中的抽象节点为 ParseTree,元对象中的所有节点均实现了此接口.该接口除了声明了 Visitor 模式所必需的 accept 方法外,也声明了一些所有节点都必须实现的公共方法.

具体节点(Concrete Node):定义了节点自己的数据结构和一些相关的操作方法,并且实现了抽象节点所声明的 accept 方法.

抽象访问者(Abstract Visitor):所有访问者的父类,声明了一个 visit 方法用于操作具体的节点数据.具体的方法操作逻辑由具体访问者实现.

具体访问者(Concrete Visitor):继承抽象访问者类或实现抽象访问者接口的实体类,然后根据自己所需要进行的操作实现或重写了抽象访问者声明的 visit 方法.

结构对象(ObjectStructure):结构对象通常通过一个集合来存储节点对象,然后提供一个更高层次的接口方便访问者对象访问每一个具体节点对象.

客户端(Client):客户端主要负责初始化节点对象数据、访问者对象数据等,然后调用结构对象里节点的 accept 方法完成相应访问者的操作逻辑.

本文采用 Visitor 模式进行设计,使得系统具备了良好的可扩展性.每当新增一个变异算子功能时只需要新增一个访问者类,然后在此类里面实现对相应节点的访问方法即可,方便了系统的扩展.同时通过将每一个访问者操作集中到一个类中,避免了一个类的多个操作分散到各个节点,降低了系统的数据和操作的耦合性.

2.4 CMuJava工具

1. CMuJava 架构

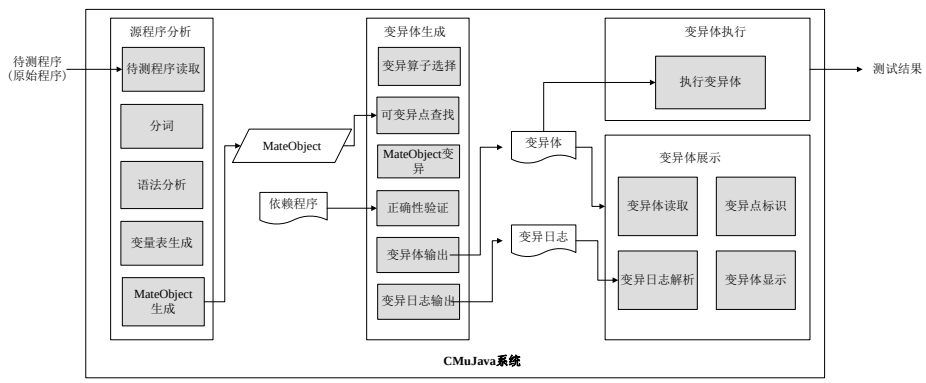


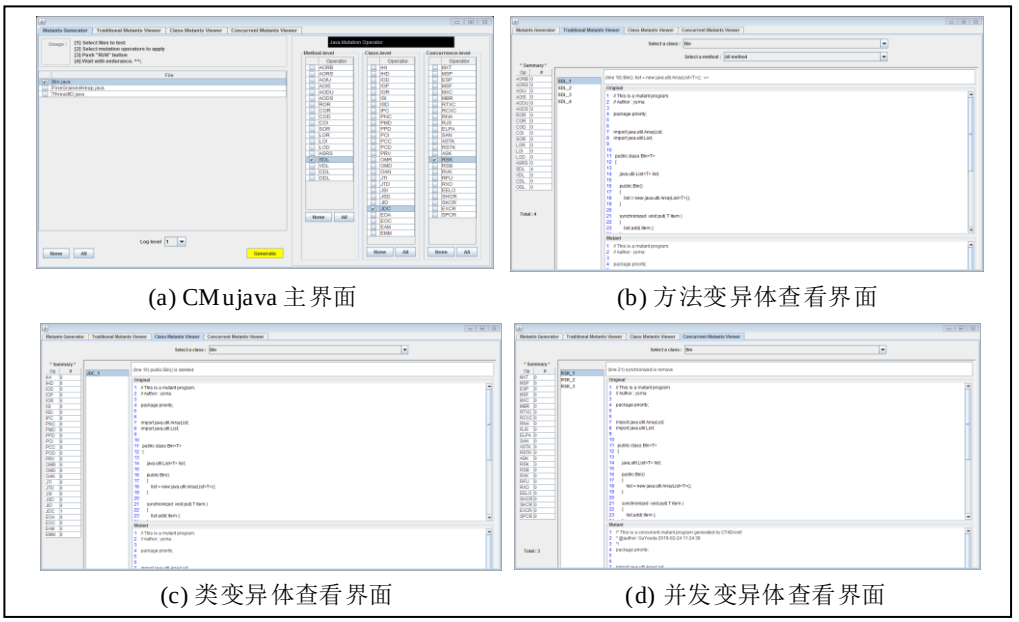
Fig.7 Structure of CMujava

图 7 CMujava 架构

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[7],设计并实现了并发变异体自动生成工具 CMuJava.该工具集成并扩展了 Mujava[4]的功能,在能生成传统的方法级别和类级别的变异体[4][11]基础上,能生成并发变异体.CMujava 的系统架构图如图 7 所示.

2. 系统演示

本节展示本文提出的支持工具 CMuJava 的研究成果.图 8 为 CMuJava 的运行时的各个界面,在主界面的最上面包含“Mutants Generator”、“Method Mutants Viewer”、“Class Mutants Viewer”和“Concurrent Mutants Viewer”四个不同的菜单栏,点击不同的菜单按钮切换到相应的功能界面.在菜单栏下面列出了工具的使用方法:首先选择待测程序,然后选择需要使用的变异算子,接下来设置日志级别,最后点击“Generate”按钮等待变异完成.



(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

如图 8(a)中所示,测试人员可以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,本文的 CMuJava 实现了其中的 23 个并发级别变异算子.在完成待测程序和变异算子选择后,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.

3 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.

3.1 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: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:分析工具在对不同规模的程序生成变异体的性能变化及工具对于提升生成并发变异体的效率的作用.

3.2 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)、LogicalOrderingAVL(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:Bin 程序是一个支持存储任意项的程序池,它具有插入数据项的 put()方法,以及删除和返回任意数据项的 get()方法,如果 Bin 程序池是空的则返回 null;ThreadID(TID)程序中提供了 get()方法,可以调用线程的标识符;FineGrainedHeap 是一个允许并发调用并行进行基准程序.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[1].

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

3.3 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,另外我们采用测试人员人工生成变异体的方式作为对照.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.

2. 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,

[1] https://logging.apache.org/log4j/2.x/

$$正确率 = \frac{生成的正确的并发变异体}{生成的所有的变异体} \times 100\% \quad (5).$$

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.

3.4 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.

2. 实验步骤

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.

4 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.

5 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.

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%.

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%

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

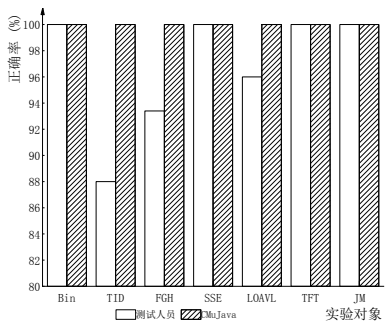


Fig.9 Correct rate comparison chart of concurrent mutants

图 9 并发变异体正确率对比图

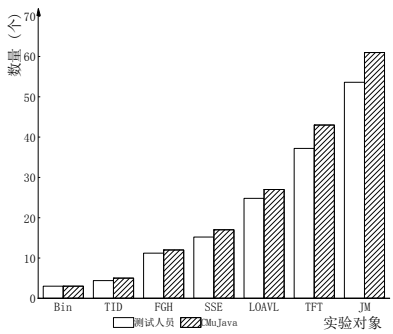


Fig.10 Number comparison chart of concurrent mutants

图 10 并发变异体的数量对比图

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.

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 CMujava 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遗漏变异体这个问题.

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 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

图 11 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,盒子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 CMuJava 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 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.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.

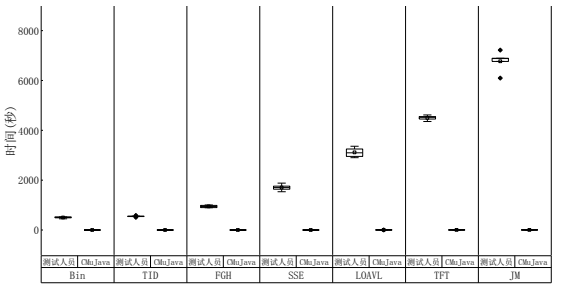


Fig.11 Boxplots of time spent on generating concurrent mutants
图 11 生成并发变异体所用时间盒图

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.

6 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[7].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.Para μ[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.

7 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 个并发程序评估并比较了 CMuJava 生成并发变异体和人工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. Paraju: 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.

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.