UNIVERSITY OF SCIENCE AND TECHNOLOGY BEIJING

SOFTWARE TESTING AND SERVICE COMPUTING LAB

并发程序变异体生成报告

姓名 ： 代贺鹏

版本 ： 2

2018 年 9 月 20 日

目录

1 内容简介

并发程序中由于子线程执行的不确定性，导致并发测试面对诸多困难。并发程序的主要特点是多个线程共同合作完成一个任务，例如：对大量数据进行排序、在大量数据中搜素某些元素等等。在这样类似的程序中很难找到一个恰当并且正确的 Oracle ，有时候即便找到了这样的 Oracle 也需要很大的代价去运用。

1998 年 Chen 提出了蜕变测试 [2],该技术提取待测程序的属性然后定义蜕变关系(MRs)。原始的测试用例和识别的蜕变关系可以用来生成衍生测试用例。然后原始测试用例和衍生测试用例分别在待测程序中执行，最后判断原始测试用例的输出以及对应的衍生测试用例的输出是否违反了蜕变关系。该技术不需要测试预期就能判定测试成功还是失败，因此极大的缓解了 Oracle 问题。

本文调查蜕变测试在并发程序中的故障检测能力。我们挑选了 5 个测试对象：SimpleLinear、SimpleTree、SequentialHeap、prioritySkip 以及 FineGrainedHeap 。这五个实验对象都实现了一个功能：并发地从大量数据中返回优先级最高的一组数据。

本文主要讨论五个程序的变异体情况。

2 变异分析

变异分析是一种基于故障的软件测试技术。变异分析的基本思想是针对某一个原始程序，运用变异算子模拟程序中的常见错误并植入源程序，这个过程称为变异生成。生成后的原始程序的错误版本成为变异体。使用若干测试用例分别在变异体以及原始程序上执行，若存在某个测试用例在变异体以及源程序上的执行结果不同，则变异体“被杀死”，即植入的故障被检测到。反之，变异体未被杀死。有些变异体虽然在语法上与原始程序不同，但是在语义上是一致的，因此没有一个测试用例能够杀死变异体，这类变异体称为等价变异体。使用某种测试技术 S 产生原始程序 P 的测试用例集 TS，可以通过原始程序及各个变异体在 TS 上的运行情况对测试技术 S 以及测试用例集 TS 进行评估。上述过程称为变异分析。变异分析中最重要的评估指标称为变异得分(Mutation Score,MS)，它是指测试用例集 TS 中能够杀死的变异体数量占变异体总量的比例。变异得分的定义如下：

$$MS(P,TS) = \frac{N_k}{N_m - N_e} \tag{1}$$

在公式 (1) 中，P 是源程序，TS 是某种技术产生的测试用例集。N_k 表示被杀死的变异体数量，N_m 表示变异体的总数量，N_e 是等价变异体的数量。

3 生成变异体的详细报告

主要介绍生成非并发变异体和并发变异体的详细过程。

3.1 非并发变异体

介绍非并发变异体（传统变异体）的生成工具、运用的变异算子以及生成变异体的详细信息。

3.1.1 变异分析工具：Major

Major 是一个变异分析框架，将变异分析划分为两个步骤：

- 在编译源程序的过程中生成变异体。
- 在变异体上执行测试用例。

在网页 http://mutation-testing.org/ 中，记录了 Major 的详细情况。

3.1.2 变异算子

在编译源程序的过程中，Major 根据变异算子生成相应的变异体。表 1 详细地描述了运用的变异算子。

Table 1: Major 涉及到的所有变异算子

变异算子	例子
AOR (Arithmetic Operator Replacement)	a + b → a - b
LOR (Logical Operator Replacement)	a ^ b → a — b
COR (Conditional Operator Replacement)	a || b → a && b
ROR (Relational Operator Replacement)	a == b → a >= b
SOR (Shift Operator Replacement)	a >> b → a << b
ORU (Operator Replacement Unary)	-a → ~ a
EVR (Expression Value Replacement)	return a → return 0
LVR (Literal Value Replacement)	0 → 1
STD (STatement Deletion)	return a → $< no - op >$

3.1.3 变异体信息

利用 Major 提供的所有变异算子，为五个源程序生成相应的变异体，表 2 详细地展示了每个程序的传统变异体信息。

Table 2: 传统的变异体信息

程序	未处理	删除执行不到的变异体	删除编译不通过的变异体	删除等价变异体
SimpleLinear	17	17	17	
SimpleTree	71	71	71	
SequentialHeap	158	106	104	
FineGrainedHeap	221	174	173	
SkipQueue	253	253	248	
总计	720	625	617	

3.2 并发变异体

主要介绍实验对象对应的并发变异体情况。

3.2.1 并发变异算子

本文对待测程序进行并发变异的依据是 Bradbury 在 [1]中提出的 24 种并发变异算子。这些变异算子可以分为 5 类：改变并发方法的参数、改变并发方法的调用、改变关键字、转换斌方法对象以及改变临界区。

待测的 5 个程序涉及到的并发机制有三种：一， synchronized 关键字修饰方法；二，原子类型；三，重入锁机制。因此根据每一个变异算子的应用情况，理论上可以作用到待测程序的有 8 种，如表 3。

Table 3: 理论上可能用到的变异算子

变异算子	变异算子说明
RCXC	移除并发机制方法的调用
SAN	将原子调用转化为非原子调用
ASTK	在非静态的 synchronized 方法前面加上 static
RSK	移除 synchronized 关键字
RFU	移除包含 unlock 语句的 finally 关键字
RXO	用另一个锁对象代替目前的锁对象
EELO	交换锁对象
ELPA	改变 lock 调用的方法

接下来对每一个程序的并发变异体详细说明。由于目前只考虑并发的移除序列中优先级前十的数据，因此只对从序列中移除优先级最高的方法应用并发变异算子。由于向序列中添加元素是顺序执行的，即便运用了并发变异算子对

添加元素的方法进行变异得到的变异体在理论上也杀不死。

3.2.2 SimpleLinear 程序

该程序的并发机制体现在 Bin 类的 3 个方法： put 、 get 、 isEmpty 。源程序在这三个方法上添加了 synchronized 关键字，使得同一时刻只能有一个线程方法这些方法。 put 方法是向序列中添加元素以及元素对应的优先级； get 方法是从序列中去除当前优先级最高的元素，并在取出之后从序列中删除；isEmpty 是判断目前的序列中是否有元素。在我们的测试用例中 isEmpty 方法没有调用的机会，因此不对其进行变异。因此运用理论上可以用 RSK 、 ASTK 变异算子进行变异。但是在运用 ASTK 进行变异时，必须对 Bin 类的成员变量 list (该成员变量存放某一个优先级对应的元素)前面加上 static 关键字变成静态成员变量(所有对象公用这个list)。我们需要每一个优先级有自己的独立的"仓库"，因此不能使用 ASTK 变异算子。

综上，运用 RSK 变异算子得到 SimpleLinear 程序的 2 个并发变异体。表 4 展示了 SimpleLinear 的并发变异体信息。

Table 4: SimpleLinear 的并发变异体

变异体	变异位置
RSK_add_1	27:void put(T item)
RSK_remove_2	31:T get()

在表 4 中，变异体的名字由 X_Y_Z 组成：X 表示变异算子的名字；Y 表示变异的方法（add方法是向序列中添加数据；remove方法是取数据）；Z 表示数量编号。变异位置由 A:B 组成：A 表示变异的具体行数；B 表示变异后的具体代码。其它程序的并发变异体信息均按照上述规则来描述。

3.2.3 SimpleTree 程序

该程序的并发机制体现在 SimpleTree 的内部类 TreeNode 的原子成员变量：counter 以及 Bin 类的 synchronized 关键字。因此可以在该程序上运用 SAN 和 RSK 变异算子，结果如表 5所示。

Table 5: SimpleTree 的并发变异体

变异体	变异位置
RSK_add_1	27:void put(T item)
RSK_remove_2	31:T get()
SAN_add_1	58:parent.counter.get();parent.counter++;
SAN_remove_2	67:if (node.counter.get() > 0) node.counter–;

3.2.4 FineGrainedHeap 程序

该程序通过实例化Lock类来实现并发，因此理论上可以运用： EELO 、 ELPA 、 RFU 、 RCXC 、 SHCR 、 EXCR 、 SKCR 算子对该程序进行变异，每个变异体的变异信息展示在表 6 中。

Table 6: FineGrainedHeap 的并发变异体

变异体	变异位置
RCXC_add_1	52:去掉heapLock.unlock
RCXC_add_2	54:去掉 heap[child].unlock
RCXC_add_3	74:去掉 heap[oldchild].unlock
RCXC_add_4	75:去掉 heap[parent].unlock
RCXC_add_5	84:去掉 heap[ROOT].unlock
RCXC_remove_1	97:去掉 heapLock.unlock
RCXC_remove_2	99、109:去掉 heap[ROOT].unlock
RCXC_remove_3	107:去掉 heap[bottom].unlock
RCXC_remove_4	120、124:去掉 heap[right].unlock
RCXC_remove_5	121、127:去掉 heap[left].unlock
RCXC_remove_6	132、139:去掉 heap[parent].unlock
RCXC_remove_7	135:去掉 heap[child].unlock
ELPA_add_1	49:heapLock.tryLock();
ELPA_add_2	49:heapLock.lockInterruptibly();
ELPA_add_3	51:heap[child].addTryLock();
ELPA_add_4	51:heap[child].addLockInterruptibly();
ELPA_add_5	57:heap[parent].addTryLock();
ELPA_add_6	57:heap[parent].addLockInterruptibly();
ELPA_add_7	58:heap[parent].addTryLock();
ELPA_add_8	58:heap[parent].addLockInterruptibly();
ELPA_add_9	79:heap[ROOT].addTryLock();
ELPA_add_10	79:heap[ROOT].addLockInterruptibly();
ELPA_remove_1	93:heapLock.tryLock();
ELPA_remove_2	93:heapLock.lockInterruptibly();
ELPA_remove_3	95:heap[bottom].removeTryLock();
ELPA_remove_4	95:heap[bottom].removeLockInterruptibly();
ELPA_remove_5	96:heap[ROOT].removeTryLock();
ELPA_remove_6	96:heap[ROOT].removeLockInterruptibly();

ELPA_remove_7	100:heap[bottom].removeTryLock();
ELPA_remove_8	100:heap[bottom].removeLockInterruptibly();
ELPA_remove_9	117:heap[left].removeTryLock();
ELPA_remove_10	117:heap[left].removeLockInterruptibly();
ELPA_remove_11	118:heap[right].removeTryLock();
ELPA_remove_12	118:heap[right].removeLockInterruptibly();
RFU_add_1	73:去掉 finally
EELO_add_1	57、58:互换两行代码
EELO_remove_1	97、98:互换两行代码
EELO_remove_2	119、120:互换两行代码
SHCR_add_1	50:放在49行的前面
SHCR_remove_1	96:放在95行前面

为了容易地生成 ELPA 类型的变异体，本文在 HeapNode 类中添加了 addTryLock() 、 addLockInterruptibly() 、 removeTryLock() 和 removeLockInterruptibly() 四个方法，具体实现如图 1。

```
public void addTryLock() { lock.tryLock(); }

public void addLockInterruptibly() throws InterruptedException {lock.lockInterruptibly();}

public void removeTryLock() { lock.tryLock(); }

public void removeLockInterruptibly() throws InterruptedException {lock.lockInterruptibly();}
```

图 1: 新增的方法

3.2.5 PrioritySkiplist 程序

该程序的并发机制是 PrioritySkiplist 类的内部类 Node 中的原子成员变量，因此可以运用SAN变异算子，具体的细节展示在表 7 中。

Table 7: PrioritySkiplist 的并发变异体

变异体	变异位置
SAN_add_1	64:if (!(pred.next[bottomLevel].getReference == succ && pred.next[bottomLevel].isMarked().equal(false))){ pred.next[bottomLevel].set(node,false);
SAN_add_2	72:if (pred.next[level].getReference == succ && pred.next[level].isMarked().equal(false)){ pred.next[level].set(node,false);

SAN_add_3	52:boolean found = addfind(node, preds, succs);
SAN_add_4	74:addfind(node, preds, succs);
SAN_remove_1	96:boolean found = removefind(node, preds, succs);
SAN_remove_2	119:removefind(node, preds, succs);
SAN_remove_3	116:if (node.next[bottomLevel].getReference() == succ && node.next[bottomLevel].isMarked().equal(false)){ node.next[bottomLevel].set(succ,true);
SAN_remove_4	132: if (curr.marked.get() == false) { curr.marked.set(true);

为了方便生成SAN类型的变异体以及执行测试，本文在 PrioritySkiplist 类中新增了两个方法： addfind() 和 removefind()。这两种方法的具体实现与该程序原有的 find() 方法一致。

3.2.6 并发变异体数量汇总

表 8 展示了所有实验对象生成的并发变异体数目情况。

Table 8: 并发变异体的数量信息

SimpleLinear	SimpleTree	FineGrainedHeap	PrioritySkiplist
2	4	40	8

参考文献

[1] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Mutation operators for concurrent java (j2se 5.0). In *The Workshop on Mutation Analysis*, page 11, 2006.

[2] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. 1998.