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 ： 1

2018 年 9 月 17 日

目录

1 内容简介 **2**

2 变异分析 **2**

3 运用的变异算子介绍 **3**

3.1 并发变异算子 3

3.1.1 SimpleLinear 程序 3

3.1.2 SimpleTree 程序 4

3.1.3 SequentialHeap 程序 4

3.1.4 PrioritySkiplist 程序 4

3.1.5 FineGrainedHeap 程序 5

3.2 传统变异算子 7

3.2.1 SimpleLinear 程序 7

3.2.2 SimpleTree 程序 7

3.2.3 SequentialHeap 程序 7

3.2.4 PrioritySkiplist 程序 8

3.2.5 FineGrainedHeap 程序 8

4 所有变异体 **9**

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 运用的变异算子介绍

3.1 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 1。

变异算子编号	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

Table 1: 理论上可能用到的变异算子

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。

3.1.1 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 1 个并发变异体。该变异算子的运用很简单这里不再放上程序进行说明。

3.1.2 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 。因此可以用 SAN 算子作用于实现移除优先级组最高的元素功能的方法：removeMin。源程序的 removeMin 方法如表 2,经 SAN 变异算子作用后的 removeMin 方法如表 3。

```
public T removeMin() {
TreeNode node = root;
while(!node.isLeaf()) {
if (node.counter.getAndDecrement() > 0 ) {
node = node.left;
} else {
node = node.right;
}
}
return node.bin.get();
}
```

Table 2: 源 SimpleTree 程序的 removeMin 方法

```
public T removeMin() {
TreeNode node = root;
while(!node.isLeaf()) {
if (node.counter.get() > 0 ) {
node.counter.set(node.counter.get() - 1);
node = node.left;
} else {
node.counter.set(node.counter.get() - 1);
node = node.right;
}
}
return node.bin.get();
}
```

Table 3: 变异 SimpleTree 程序的 removeMin 方法

3.1.3 SequentialHeap 程序

我们在实验的过程中发现在一些并发情况下源程序存在缺陷。因此我们在 removeMin 方法前面加上了 synchronized 关键字(由于该方法调用了 swap 方法，因此在swap方法前面也加上了 synchronized 关键字)。由此理论上就可以用 RSK 以及 ASTK 变异算子进行变异。但是用 ASTK 算子要在该类实现的接口上进行修改还需要对该类的其它成员变量进行修改，因此我们没有运用ASTK算子作用源程序。运用RSK分别作用 removeMin 、 swap 之后得到两个变异体：RSK_1、RSK_2。由于该变异算子的运用方式很简单，这里不在现实具体程序。

3.1.4 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。该类中涉及到对原子变成尽享操作的方法有：add 、 remove 、 findAndMarkMin 、 find 。其中 add 方法我们暂时不考虑它的并发； remove 方法是在物理上删除已经移除的元素，不管该方法有没有故障都不影响最后的结果，因此不对该方法进行变异。 findAndMarkMin 方法实现了

发现并返回优先级最高的元素的功能，因此需要对该方法运用 SAN 变异算子；find 方法主要被add以及remove调用，因此该方法也不涉及到并发变异体。

综上，对PrioritySkiplist 程序中的 findAndMarkMin 方法运用 SAN 变异算子得到一个变异体。

```
public Node<T> findAndMarkMin() {
Node<T> curr = null;
curr = head.next[0].getReference();
while (curr != tail) {
if (!curr.marked.get()) {
if (curr.marked.compareAndSet(false, true)) {
return curr;
} else {
curr = curr.next[0].getReference();
}
}
}
return null;
}
```

Table 4: 源 PrioritySkiplist 程序的 findAndMarkMin 方法

```
public Node<T> findAndMarkMin() {
Node<T> curr = null;
curr = head.next[0].getReference();
while (curr != tail) {
if (!curr.marked.get()) {
if (curr.marked.get() == false) {
curr.marked.set(true);
return curr;
} else {
curr = curr.next[0].getReference();
}
}
}
return null;
}
```

Table 5: 变异 PrioritySkiplist 程序的 findAndMarkMin 方法

3.1.5 FineGrainedHeap 程序

该程序通过调用lock方法实现并发，因此理论上可以运用： RXO 、 EELO 、 ELPA 、 RFU 、 RCXC 算子对该程序进行变异。同上只对 removeMin 方法进行变异，removeMin 代码如图 1。

从图中可以看出 RFU 算子的作用机制不适用与该方法。EELO 算子可以产生 3 个变异体：一，在 1 模块将 heapLock.lock(); 与 heap[ROOT].lock(); 进行交换；二，在 3 模块中可以用 heap[ROOT].lock(); 交换 heap[bottom].lock(); 然而不能用 heap[ROOT].lock(); 与 heap[bottom].lock(); 交换。原因是在原始 heap[ROOT].lock(); 的位置变量 bottom 还没有声明；三，在 9 模块将 heap[left].lock(); 、 heap[right].lock(); 交换位置。没有用 heapLock.lock(); 、 heap[ROOT].lock(); 和 heap[bottom].lock(); 分别交换 heap[left].lock(); 和 heap[right].lock(); 中任何一个的原因是当交换的时候由于 left 、 right 变量没有声明会报错。故此变异算子可以产生 3 个变异体。

ELPA 算子替换 lock 方法，本文用 trylock 方法代替原有的 lock 方法：一，模块 1 中的 heapLock.lock() 替换为 heapLock.tryLock() ； 二，该类的内部类 HeapNode 中的方法由表 6中代码换成表 7中代码。故此变异算子可以产生2个变异体。

RXO 算子用另外一个锁代替某一个锁。可以用 heapLock.lock(); 与 heap[ROOT].lock(); 分别代替模块 1 中的 heap[bottom].lock();。可以产生 2 个变异体。不用其它锁代替 heap[left].lock(); 和 heap[right].lock(); 的原因是：代替之后由于上文没有出现该锁对象导致 heap[right].unlock(); 和 heap[left].unlock(); 成为无意义的语句。

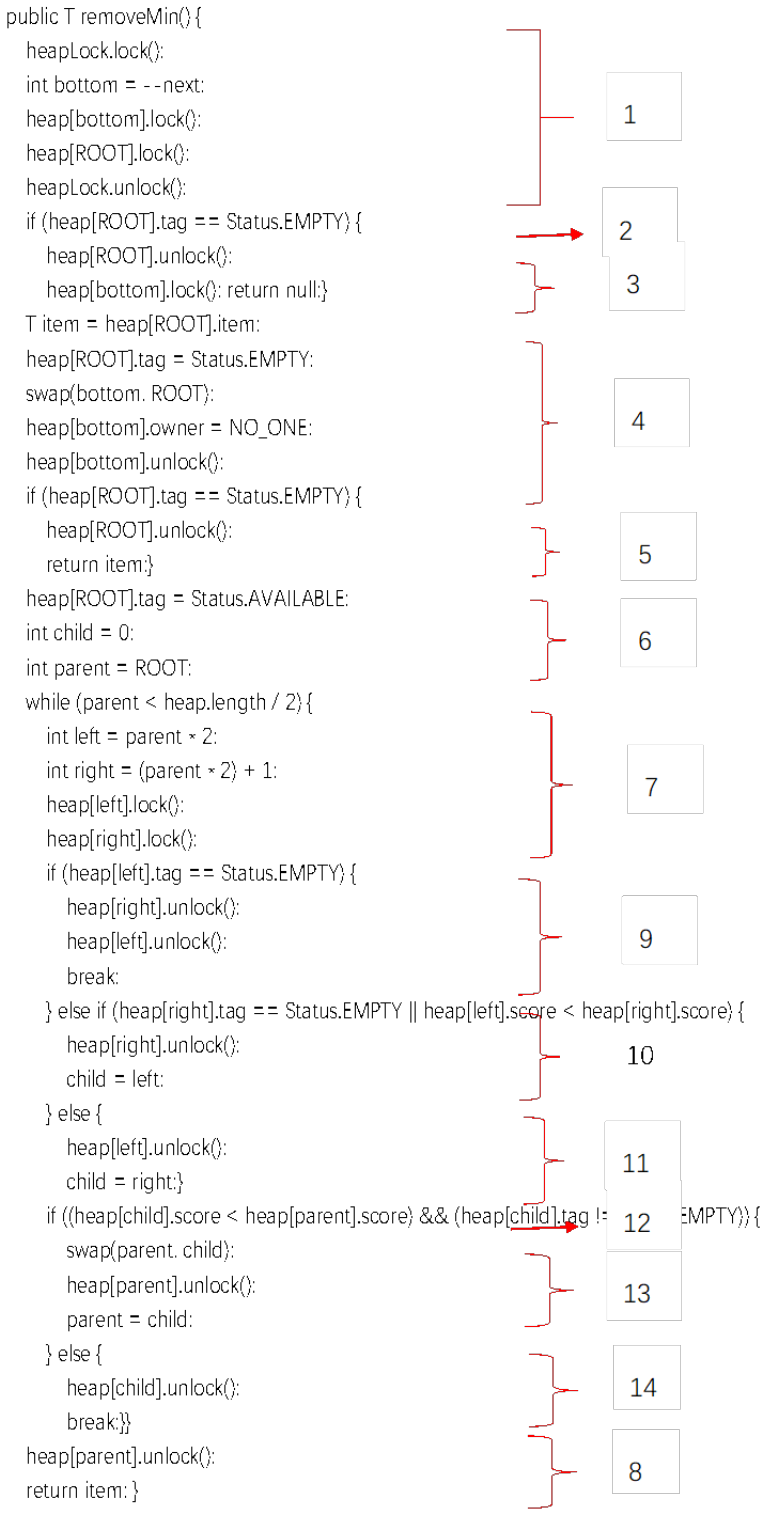

图 1: FineGrainedHeap 程序 removeMin 方法

RCXC 算子移除 unlock 方法的调用， removeMin 方法一共出现了 11 个 unlock 方法的调用，分别将之移除可以得到 11 个变异体。该算子的运用方式很简单这里不一一说明。

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

Table 6: 源 FineGrained-Heap 程序的lock()方法

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

Table 7: 变异 FineGrained-Heap 程序的lock()方法

3.2 传统变异算子

由于我们的研究对象涉及的类不复杂并且与其它类的依赖很小，因此不涉及类级别的变异算子。我们对待测程序进行变异的依据是 Ma 在 [3]中涉及的16种变异算子，如表 8所示。所有的变异算子可以分为 7 类：算术算子、关系算子、条件算子、位运算、逻辑运算、赋值算子和删除算子。将这些变异算子所用到 5 个待测程序之后，去掉不能是程序正常执行的变异体以及等价变异体之后，每一个实验对象由传统变异算子得到的变异体如表 9所示。

我们的研究中所有的变异体都是由 Mujava [4]产生的。在原始程序中会有一些方法，我们的测试用例覆盖不到或者该方法不管对错对输出结果没有影响，因此不对这些方法进行变异。接下来对每一个程序的传统变异算子得到的变异体做详细说明。

3.2.1 SimpleLinear 程序

该类一共有三个函数：构造函数、 add 、 removeMin 。这三个函数我们设计的测试用例都能够覆盖到。因此从生成的所有变异体中剔除使程序不能正常执行的变异体以及等价变异体后剩余 13 个。

3.2.2 SimpleTree 程序

该类一共有 4 个函数：构造函数、 add 、 removeMin 、 buildTree 。这四个函数我们设计的测试用例都能够覆盖到。因此从生成的所有变异体中剔除使程序不能正常执行的变异体以及等价变异体之后剩余 28 个。

3.2.3 SequentialHeap 程序

该类一共有 6 个函数：构造函数、 add 、 removeMin 、 getMin 、 swap 、 isEmpty 、 sanityCheck 。我们的测试用例能够覆盖到的方法为：构造函数、 add 、 removeMin 、 swap 。因此从所有的变异体中挑出这个四个函数的变异体，然后去掉是程序不能正常执行的变异体以及等价变异体之后剩余244个变异体。

category	Operator	Description
Arithmetic	AORB	Arithmetic: Operator Replacement(binary)
	AORU	Arithmetic Operator Replacement (unary)
	AORS	Arithmetic Operator Replacement (short-cut)
	AOIU	Arithmetic Operator Insertion (unary)
	AOIS	Arithmetic Operator Insertion (short-cut)
	AODU	Arithmetic Operator Deletion (unary)
	AODS	Arithmetic Operator Deletion (short-cut)
Relational	ROR	Relational Operator Replacement
Conditional	COR	Conditional Operator Replacement
	COI	Conditional Operator Insertion
	COD	Conditional Operator Deletion
Shift	SOR	Shift Operator Replacement
Logical	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment	ASRS	Assignment Operator Replacement
Deletion	SDL	Statement Delection
	VDL	Variable Deletion
	CDL	Constant Deletion
	ODL	Operator Deletion

Table 8: 传统变异算子

3.2.4 PrioritySkiplist 程序

该类一共有 7 个函数：构造函数、 randomLevel 、 add 、 remove 、 findAndMarkMin 、 find 、内部类的函数。我们的测试用例可以覆盖的函数为：：构造函数、 randomLevel 、 add 、 remove 、 findAndMarkMin 、 find 、内部类的函数。由于 Mujava 不能识别 randomLevel 函数中的>>>符号，因此不对该函数进行变异。remove 函数是物理上删除某一个元素，不管这个函数实现的对不对，只要在逻辑上删除了已经取到的元素对结果的输出都没有影响，因此对该函数产生的变异体均为等价变异体。因此不对其进行变异。

通过上面的阐述，从所有的变异体中去掉使程序不能正常执行的变异体以及等价变异体之后剩余24个变异体。

3.2.5 FineGrainedHeap 程序

该类 6 个函数：构造函数、 add 、 removeMin 、 swap 、 sanityCheck 、内部类函数。我们的测试用例可以覆盖构造函数、 add 、 removeMin 、 swap 、

Operator	SimpleLinear	SimpleTree	SequentialHeap	FineGrainedHeap	SkipQueue
AORB	-	3	9	7	-
AORU	-	-	-	-	-
AORS	-	-	1	-	-
AOIU	1	-	10	9	2
AOIS	3	-	72	28	-
AODU	-	-	-	-	-
AODS	-	-	-	-	-
ROR	4	7	37	24	6
COR	-	-	2	4	-
COI	2	3	9	11	3
COD	-	-	-	-	1
SOR	-	-	-	-	-
LOR	-	-	-	-	-
LOI	-	-	14	11	2
LOD	-	-	-	-	-
ASRS	-	-	-	-	-
SDL	3	7	19	21	4
VDL	-	-	1	-	-
CDL	-	3	5	2	-
ODL	-	4	8	7	1
总计	13	27	187	124	19

Table 9: 传统变异算子的应用情况

内部类函数。从这个函数中生成的变异体去掉使程序不能正常执行的变异体以及等价变异体之后剩余142个变异体。

4 所有变异体

我们研究的五个实验对象的所有变异体情况如表 11，并且可以将这些变异体进行分组如表 10。

Category	Operators	Programs				
		SimpleLinear	SimpleTree	SequentialHeap	FineGrainedHeap	SkipQueue
Category1	AORB,AORS,AOIU,AOIS	4	3	92	44	2
Category2	ROR	4	7	37	24	6
Category3	COR,COI,COD	2	3	11	15	4
Category4	LOI	0	0	14	11	2
Category5	SDL,VDL,CDL,ODL	3	14	33	30	5
Category6	RCXC,SAN,ELPA	0	1	0	13	1
Category7	RSK	1	0	2	0	0
Category8	RXO,EELO	0	0	0	5	0

Table 10: 所有变异体

Operator	SimpleLinear	SimpleTree	SequentialHeap	FineGrainedHeap	SkipQueue
AORB	-	3	9	7	-
AORU	-	-	-	-	-
AORS	-	-	1	-	-
AOIU	1	-	10	9	2
AOIS	3	-	72	28	-
AODU	-	-	-	-	-
AODS	-	-	-	-	-
ROR	4	7	37	24	6
COR	-	-	2	4	-
COI	2	3	9	11	3
COD	-	-	-	-	1
SOR	-	-	-	-	-
LOR	-	-	-	-	-
LOI	-	-	14	11	2
LOD	-	-	-	-	-
ASRS	-	-	-	-	-
SDL	3	7	19	21	4
VDL	-	-	1	-	-
CDL	-	3	5	2	-
ODL	-	4	8	7	1
并发算子	1	1	2	18	1
总计	14	28	189	142	20

Table 11: 所有变异体

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.

[3] Yu-seung Ma. Description of method-level mutation operators for java. 12 2017.

[4] Jeff Offutt, Yu Seung Ma, and Yong Rae Kwon. An experimental mutation system for java. *Acm Sigsoft Software Engineering Notes*, 29(5):1–4, 2004.