DOI: 10.13718/j.cnki.xdzk.2015.04.010

基于 AdditionalUL 策略的回归测试用例优先级排序①

唐海鹏[1]，　丁晓明[1,2]

1. 西南大学 计算机与信息科学学院，重庆 400715；2. 重庆市软件评测中心有限公司，重庆 400715

摘要：Additional 策略被广泛应用于测试用例的优先级排序，其排序结果与其他方法相比，具有较好的错误检测速率. 但当遇到具有相同代码覆盖率的测试用例时，Additional 策略采用随机选择方式，该方式降低了排序效果. 基于此提出一种新的带使用标签的 Additional 策略，简称 AdditionalUL 策略. 新策略根据捕捉到的测试用例的执行信息为测试用例添加标签，设定测试用例的优先级，以此优化排序效果. 分别使用 Additional 策略与 AdditionalUL 策略对多组程序的测试用例集排序，并利用 APFD 评测指标衡量排序结果，实验表明，改进后的方法能够提高 Additional 策略的排序效果，具有更高的错误检测速率.

关　键　词：Additional 策略；AdditionalUL 策略；回归测试；测试用例优先级排序；标签

中图分类号：TP311　　**文献标志码**：A　　**文章编号**：1673-9868(2015)04-0055-07

回归测试是软件测试过程中一个较为频繁的活动. 回归测试是对软件完成一轮测试，将发现的问题修复后，为验证软件修复的正确性，重新对软件进行的测试. 然而，随着软件规模越来越庞大，受时间与成本的制约，无法对软件进行彻底的回归测试[1]. 因此需要对测试用例集进行优化.

根据软件开发流程，对测试用例优先级排序的研究主要分为 3 类[2]：需求、模型、代码. 基于需求的排序技术主要考虑与需求相关的因素，如实现复杂情况、需求变更情况、需求覆盖程度等，将各个因素量化之后，求出整体的加权值对测试用例排序[3-5]；基于模型的排序技术主要利用状态图、顺序图、扩展有限状态机、规格描述语言等表示系统行为，通过对比系统修改前后模型的差异，收集测试用例对模型差异的覆盖情况，在此基础上，确定测试用例的执行顺序[6-9]；基于代码的排序技术主要基于测试用例对代码的覆盖情况，根据覆盖程度对测试用例排序[10]. 其中，基于代码的排序技术具有易量化且较好实现的优点，是回归测试用例优先级排序研究中的一个热点，具体的排序策略主要有 Total 策略、Additional 策略、2-Optimal 策略、爬山法、遗传算法等[10-13]，其中 Additional 策略的有效性较为突出.

Additional 策略(附加策略)是基于代码覆盖技术的典型代表，优先选择覆盖率高的测试用例并动态考虑测试用例每次执行后的状况，使得回归测试能够尽快达到较高的覆盖率，提高发现错误的速率[14]，在测试用例优先级排序方面具有优势. 然而，当遇到多个测试用例具有相同覆盖率时，Additional 策略随机从中选择一个测试用例，该方式降低了测试用例的使用效率. 通过文献检索，发现目前针对 Additional 策略中存在的随机选择性，并没有正面的改进方法. 基于此，本文提出一种新的排序策略 AdditionalUL 策略(Additional with Usage Label，带使用标签的附加策略)，该策略通过为测试用例添加标签值决定具有相同覆盖率的测试用例的执行顺序.

1　Additional 策略

Additional 策略由 Rothermel 等人[10]提出，根据测试用例对程序代码的历史覆盖情况，利用贪婪法排

① 收稿日期：2014-10-11
基金项目：国家自然科学基金(61003203)；重庆市科技攻关计划(CSTC2012GGB00004).
作者简介：唐海鹏(1989-)，男，江苏盐城人，硕士研究生，主要从事软件测试方面的研究.
通信作者：丁晓明，副教授.

序，使得具有较高覆盖率的测试用例优先执行. 同时引入反馈机制，从整体角度出发，认为当一部分程序实体被已执行测试用例覆盖后，对剩余测试用例进行排序时则不需要考虑该部分程序实体的覆盖. 下面对Additional策略确定测试用例执行顺序的具体步骤进行详细介绍.

步骤1：统计各个测试用例对程序实体的覆盖情况；

步骤2：从测试用例集中选出1个覆盖率最高的测试用例(如果存在多个覆盖率最高的测试用例，则随机从中选择1个)，记录该测试用例覆盖的程序代码；

步骤3：当程序代码全部被覆盖或剩余的测试用例无法覆盖尚未覆盖的程序代码时，第1步排序终止，然后重置已被覆盖的程序代码，转步骤4，否则，直接转步骤4；

步骤4：更新剩余测试用例对剩余代码的覆盖情况；

步骤5：以此重复步骤2至步骤4，直到所有的测试用例排序完毕则结束.

以软件测试中最常用的验证程序——判断三角形类型为例，介绍Additional策略的排序思想. 待测程序如图1所示，该程序共有10行语句，图中每行语句前均标有行号. 其中在判断是否为非三角形的if判断语句(第4行语句)处人工植入了1个错误.

```
    void main()
    {
1.      float a,b,c;
2.      printf("输入数字 a、b、c：");
3.      scanf("%f%f%f",&a,&b,&c);
4.      if(a+b<c||a+c<=b||b+c<=a)
        //错误，正确应为：if(a+b<=c||a+c<=b||b+c<=a)
5.          printf("非三角形");
6.      else if(a==b&&a==c&&b==c)
7.          printf("等边三角形");
8.      else if(a==b||b==c||a==c)
9.          printf("等腰三角形");
        else
10.         printf("其他三角形");
    }
```

图1 判断三角形类型待测程序

从等价类划分、容错性等角度设计了10组测试用例，如表1所示，其中$T2$,$T3$是能够发现程序中语句错误的测试用例. 表1中最后1列代表测试用例的执行结果. 测试用例对程序语句的覆盖情况如表2所示，其中第1列为语句行号，第1行为测试用例，表格中1表示该行的语句被该列的测试用例覆盖，0表示该行的语句未被该列的测试用例覆盖，每列最后1行由该列数字相加所得，代表该列测试用例覆盖语句的总数.

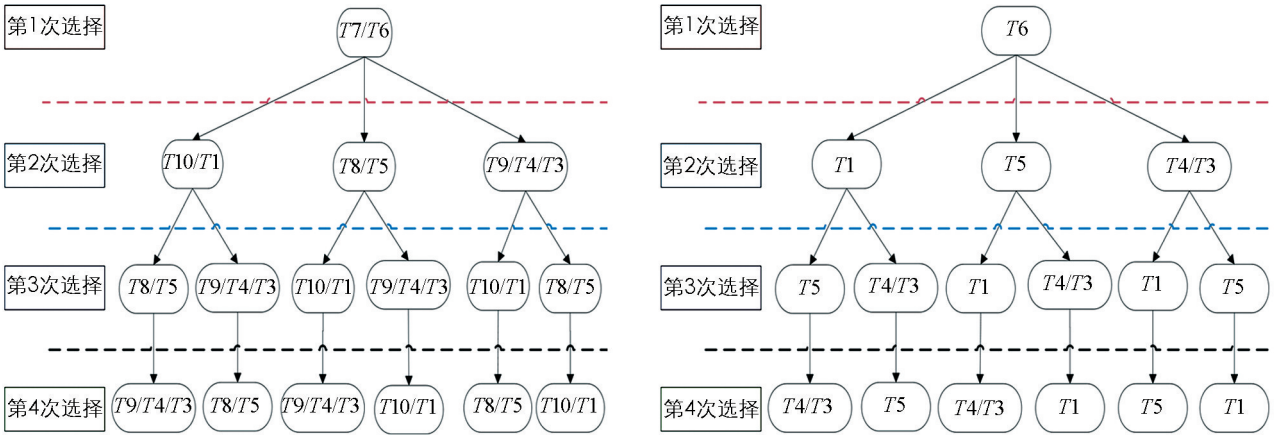
表1 判断三角形类型程序测试用例

	a	b	c	测试用例结果		a	b	c	测试用例结果
$T1$	−1	−1	0	非三角形	$T6$	3	4	5	其他三角形
$T2$	s	3	2	error	$T7$	4	3	5	其他三角形
$T3$	2	2	4	等腰三角形	$T8$	1	1	1	等边三角形
$T4$	3	3	4	等腰三角形	$T9$	3	2	2	等腰三角形
$T5$	2	2	2	等边三角形	$T10$	3	2	1	非三角形

表2 测试用例对语句覆盖情况

	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$	$T7$	$T8$	$T9$	$T10$
1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1
4	1	0	1	1	1	1	1	1	1	1
5	1	0	0	0	0	0	0	0	0	1
6	0	0	1	1	1	1	1	1	1	0
7	0	0	0	0	1	0	0	1	0	0
8	0	0	1	1	0	1	1	0	1	0
9	0	0	1	1	0	0	0	0	1	0
10	0	0	0	0	0	1	1	0	0	0
总计	5	3	7	7	6	7	7	6	7	5

按照 Additional 策略对测试用例 $T1\sim T10$ 进行排序，由表 2 可得，初始时覆盖率最大的测试用例为 $T3,T4,T6,T7$ 和 $T9$. 首先，随机从中选择 1 个测试用例，如果选择 $T7$ 或 $T6$，则对剩余测试用例覆盖剩余代码的情况更新后，第 2 次具有相同最高覆盖率的测试用例为 $T1,T3,T4,T5,T8,T9$ 和 $T10$；再次从这 7 个测试用例中随机选择 1 个，如果选择 $T10$ 或 $T1$，则第 3 次具有相同最高覆盖率的测试用例为 $T3,T4,T5,T8$ 和 $T9$；然后从这 5 个测试中随机选择 1 个，如果选择 $T8$ 或 $T5$，则第 4 次可供选择的测试用例为 $T3,T4,T9$. 随机从这 3 个测试用例中选择 1 个，会出现剩余的测试用例无法覆盖尚未覆盖的程序代码的情况，第 1 步排序终止. 上述排序过程及其它的测试用例组合可用一棵生成树表示，如图 2(a)所示. 假设第 1 步选择的测试用例顺序依次是 $T7,T10,T8,T9$，则再次排序时可供选择的组合同样用生成树表示，见图 2(b). 假设第 2 步选择的测试用例顺序依次是 $T6,T1,T5,T4$，则剩余的 2 个测试用例为 $T2$ 和 $T3$. 由表 2 得测试用例 $T2,T3$ 的覆盖率分别为 3,7，因此，最终的测试用例顺序为：$T7,T10,T8,T9,T6,T1,T5,T4,T3,T2$. 观察该测试用例的排序可发现，能够发现错误的测试用例 $T3$，$T2$ 被排在 10 个测试用例中的第 9 和第 10 位，即需要运行 90%的测试用例后才可发现待测程序的错误，该效率是很低的.



(a) 第1步排序组合　　(b) 第2步排序组合

图 2　Additional 策略 2 步排序可供选择的测试用例序列组合

因此，当采用随机的方式来处理多个具有相同覆盖率的测试用例时，发现错误的测试用例可能未被优先选择，甚至有可能把发现错误的测试用例排到最后，降低了错误检测速率. 而具有相同覆盖率的测试用例在选择过程中是普遍存在的. 因此应制定一种比较合理的选择策略，使能发现错误的测试用例被较早执行. 基于此，本文提出 AdditionalUL 策略.

2　AdditionalUL 策略

根据回归测试在选择具有相同覆盖率的测试用例时可能遇到的几种情况，将测试用例分为如下 4 类：

① 该测试用例能发现错误且被选中：表示该测试用例在上一轮测试过程中，已被执行而且发现程序中的错误.

② 该测试用例未能发现错误且被选中：表示该测试用例在上一轮测试过程中，已被执行但未发现程序中的错误.

③ 该测试用例能发现错误且未被选中：表示该测试用例在上一轮测试过程中，未执行，但在之前执行时发现过错误.

④ 该测试用例未能发现错误且未被选中：表示该测试用例在上一轮测试过程中，未执行，且在之前执行时未发现错误.

其中，③,④情况只适用于上一轮回归测试时因某种原因而未执行的测试用例. 例如，要求当覆盖率达到最大时就停止测试；要求当测试一定数量的测试用例时，就停止测试.

当对具有相同覆盖率的测试用例进行优先级排序时，能发现错误的测试用例要提高它们的优先级，对于未能发现错误的测试用例要降低其优先级. 为保证每个测试用例均能被最优执行，对每类情况设置标签

值，如表3所示. 其中，各测试用例的标签值默认为－1，$X0$ 中 X 表示该测试用例已经执行的次数($X0$ 是一个数，是10的倍数)，从上到下测试用例的优先级依次降低. 通过比较各个测试用例的标签值对测试用例进行排序.

表3 测试用例使用标识及对应含义

用例使用标识	含 义
3	该测试用例能发现错误且被选中
2	该测试用例能发现错误且未被选中
1	该测试用例未能发现错误且未被选中
$X0$	该测试用例未能发现错误且被选中
－1	测试用例标签默认值

同样以图1判断三角形类型的程序为例，详述改进策略的执行情况，并与传统 Additional 策略排序结果进行比较. 当将程序中的错误修复之后进行回归测试时，根据表3制定的规则. $T1$，$T4$～$T10$ 均未发现错误且被选中，设置其标识值为10，$T2$，$T3$ 能发现错误且被选中，因此 $T2$，$T3$ 标记为3. 对具有相同覆盖率的 $T3$，$T4$，$T6$，$T7$，$T9$ 根据标识值进行排序，$T3$ 应排在最前. 最终得出的排序结果之一如下：$T3$，$T10$，$T5$，$T6$，$T4$，$T1$，$T8$，$T9$，$T7$，$T2$. 优化后将可以检测到错误的测试用例 $T3$ 排在了第1位，即优化后只需运行一个测试用例，就可以检查出程序中的错误，加快了检验错误修复情况的速度.

而且改进后的策略在回归测试时可以充分考虑时间和成本因素，如设置当覆盖率达到最大或 N 次使得覆盖率达到最大时则停止测试，设置测试一定数量的测试用例时则停止测试. 然后进行错误修复的工作，而不必等到所有的测试用例都执行完才开始修复. 根据规则，对修复后的程序进行再次回归时，依然会优先选择未被使用的测试用例.

然后利用 Rothermel 等人提出的 APFD(Average Percentage of Fault Detection，缺陷检测加权平均值)评价指标[10]对改进前后排序算法的效率进行评估，APFD 具体含义为：计算给定的测试序列在测试过程中检测到错误的平均累计值的比例，用 V_{APFD} 来表示，取值为(0，1)，而且值越高，表示对应的测试用例集发现错误的速度越快. 假设测试用例集 T，其中含有 n 个测试用例，m 个缺陷，给定一个测试用例执行顺序，其中 TF_i 表示首个可检测到第 i 个缺陷的测试用例在该执行顺序中所处的位置，则 APFD 的计算公式如下：

$$V_{\mathrm{APFD}}=1-\frac{TF_1+TF_2+\cdots+TF_m}{n\times m}+\frac{1}{2n} \tag{1}$$

对于判断三角形类型测试程序的10组测试用例，Additional 策略对其排序的组合情况有1 728种，可以计算所有组合的 APFD 值最小为0.1，最大值为0.5. 利用 AdditionalUL 策略排序结果的 APFD 的值均为0.5. 该结论表明，带使用标签的 Additional 策略优于传统的 Additional 策略，具有较高的错误检测速率.

3 实验分析

本文测试了6组实验程序，所有实验程序均来源于 SIR 网站[15]，该网站提供了程序源代码、带有错误的多个程序版本及相应的测试用例集. 从中选取6个测试程序 printtokens，printtokens2，schedule，schedule2，space，airline，其中前5个为C程序，他们的规模、版本信息、测试用例数量、错误个数如表4所示. 其中，代码行代表各程序正确版本的总代码行数，版本数包括了正确版本和错误版本，错误个数为该程序所有版本中总错误数之和. 第6个程序 airline 是 JAVA 程序，该程序共有31行代码，由于 SIR 网站只提供了一组发现错误的测试用例，因此，我们首先按照等价类划分的方法，结合容错性将测试用例扩充至16组，具体的测试用例如表5所示，测试用例用 $Ty(y\in\{1,2,\cdots,16\})$ 标记. 其中，$T3$，$T6$，$T8$，$T9$，$T11$，$T12$，$T13$，$T16$ 用例均抛出异常，$T10$ 虽未抛出异常，但 cushion 不能为负数，$T5$ 有时正常执行，有时抛出异常，故发现错误的测试用例有 $T3$，$T5$，$T6$，$T8$，$T9$，$T10$，$T11$，$T12$，$T13$，$T16$，其他测试用例未发现错误. 根据具体的程序错误，对发现错误的测试用例归类，其中 $T6$，$T10$，$T13$ 为不符合事实错误(cushion 和 numberThread 不能为负数)，$T3$，$T5$，$T8$，$T9$，$T16$ 使得系统抛出异常，$T11$，$T12$ 为参数类型不符.

表 4　程序信息

程　序　名	代码行	版本数	测试用例数量	错误个数
printtokens(程序 1)	726	8	4 130	7
printtokens2(程序 2)	570	11	4 115	10
schedule(程序 3)	412	10	2 650	9
schedule2(程序 4)	374	11	2 710	10
space(程序 5)	9 564	36	13 585	35

表 5　测试用例集

	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$	$T7$	$T8$
numberThread(线程数)	10	2	2	0	10	−1	1	3
cushion(缓冲数)		1	2	0	3	2	1	10
	$T9$	$T10$	$T11$	$T12$	$T13$	$T14$	$T15$	$T16$
numberThread(线程数)	0	2	s	1	1	1	2	1
cushion(缓冲数)	2	−1	1	s	−1	0	0	2

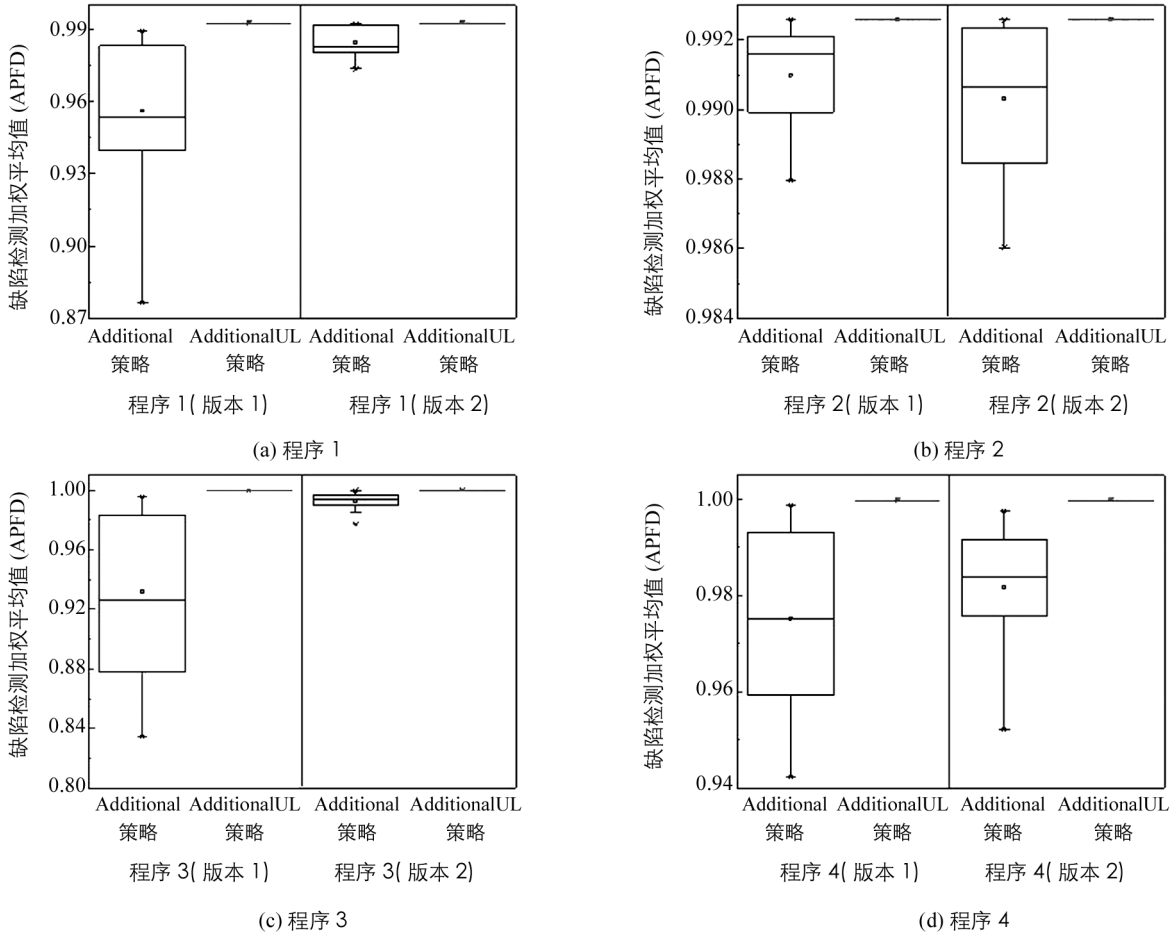


图 3　Additional 策略和 AdditionalUL 策略分别对程序 1～4 中 2 个版本排序结果的 APFD 值比较

利用覆盖率统计工具 lcov 统计 5 个 C 程序的测试用例对应的代码覆盖情况，利用覆盖率统计工具 cobecover 统计 JAVA 程序的测试用例对应的代码覆盖情况. 结合代码覆盖及发现错误情况，分别使用 Additional 策略和 AdditionalUL 策略对各程序的测试用例排序，比较 2 种排序策略所得测试用例序列的 APFD 的值. 每个 C 程序选取 2 个发现错误的版本，并对每个版本的测试用例集及 airline 的测试用例集运用这 2 种策略排序 10 次，对结果的 APFD 值画出箱式图，如图 3 和图 4.

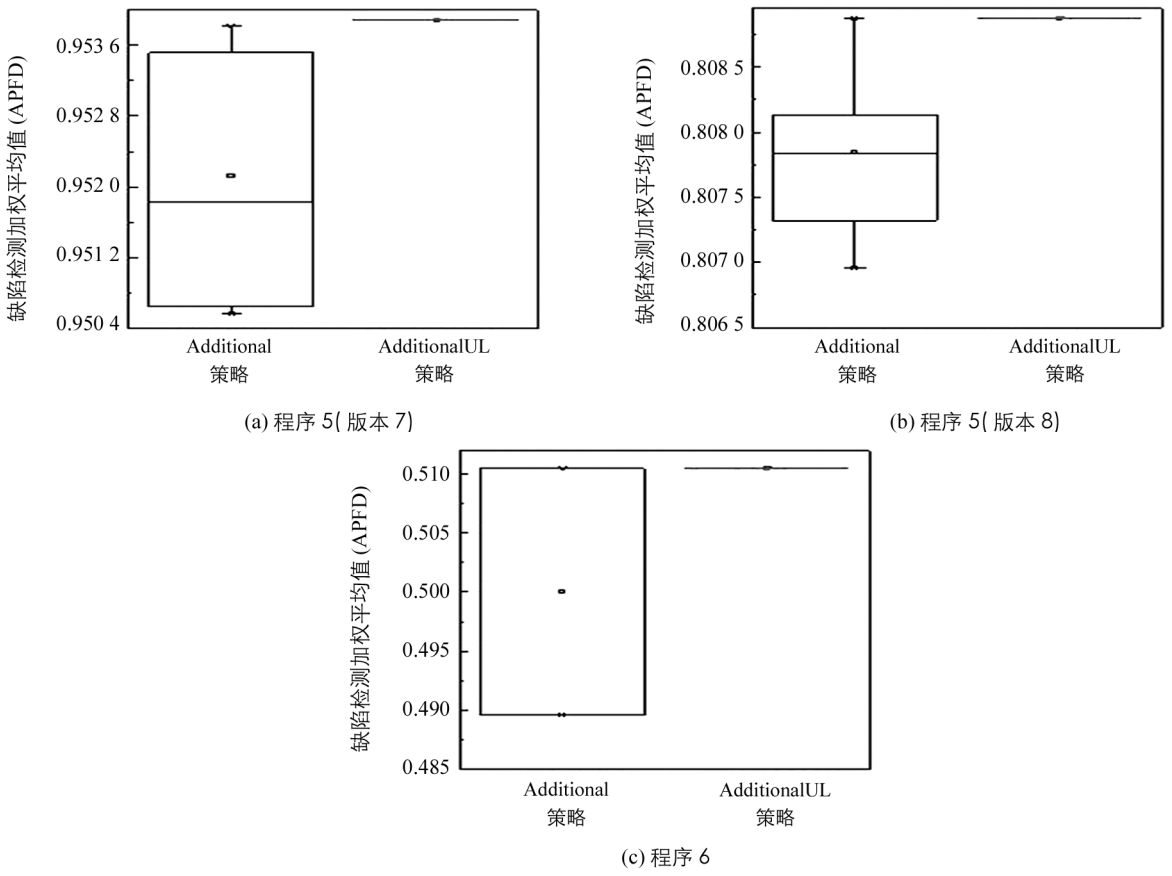


图4 Additional策略和AdditionalUL策略分别对程序5～6中2个策略排序结果的APFD值比较

从图3和图4比较结果可以看出Additional策略排序结果的APFD值波动范围较大. 以图3(a)中的版本1为例，其最大值可以达到0.983，最小值却为0.876. 这是因为Additional策略的随机选择造成的. 而AdditionalUL策略排序结果的APFD值十分稳定，在统计每个程序的每个版本的10次结果中所有的值都一样，且该值不小于Additional策略排序结果中APFD值的最大值. 例如图3(a)中，AdditionalUL策略排序结果的APFD值恒为0.993，并且大于Additional排序结果的APFD值的最大值.

通过上述实验，验证了AdditionalUL策略可以有效改进Additional策略的随机选择性，提高Additional策略的排序效果，在回归测试时可以有效提高错误的检错速率.

4 小 结

针对Additional策略随机选择的不足，提出利用带使用标签的附加策略的思想对其优化，帮助Additional策略在遇到多个具有相同覆盖率的测试用例时，根据测试用例发现的错误及使用情况，进一步动态调整测试用例的执行顺序，使其保持较高的错误发现速率. 经过实验验证，改进后的方法能够使Additional策略选择最优的测试用例序列进行回归测试，并且提高Additional策略排序结果的稳定性，在很大程度上提高测试效率，节约测试成本.

参考文献：

[1] YOO S, HARMAN M. Regression Testing Minimization, Selection and Prioritization: a Survey [J]. Software Testing, Verification and Reliability, 2012, 22(2): 67－120.

[2] 陈 翔，陈继红，鞠小林，等. 回归测试中的测试用例优先排序技术述评 [J]. 软件学报，2013，24(8)：1695－1712.

[3] SRIKANTH H, WILLIAMS L. On the Economics of Requirements-Based Test Case Prioritization [J]. Proceedings of the Seventh International Workshop on Economics-Driven Software Engineering Research. New York: ACM, 2005, 30(4): 1－3.

[4] SALEHIE M, LI S, TAHVILDARI L, et al. Prioritizing Requirements-Based Regression Test Cases: A Goal-Driven Practice [C]// 2011 15th European Conference on Software Maintenance and Reengineering. Oldenburg, Germany: IEEE, 2011: 329－332.
[5] 李华莹，柴丽雅. 由需求得到回归测试用例排序技术 [J]. 计算机技术与发展，2013，23(11)：70－73.
[6] KOREL B, TAHAT L H, HARMAN M. Test Prioritization Using System Models [C]// Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). Budapest, Hungary: IEEE, 2005: 559－568.
[7] KOREL B, KOUTSOGIANNAKIS G, TAHAT L H. Model-Based Test Prioritization Heuristic Methods and Their Evaluation [C]// Proceedings of the 3rd International Workshop on Advances in Model-Based Testing. New York: ACM, 2007: 34－43.
[8] KOREL B, KOUTSOGIANNAKIS G, TAHAT L H. Application of System Models in Regression Test Suite Prioritization [C]// IEEE International Conference on Software Maintenance, ICSM 2008. Beijing, China: IEEE, 2008: 247－256.
[9] KUNDU D, SARMA M, SAMANTA D, et al. System Testing for Object-Oriented Systems with Test Case Prioritization [J]. Software Testing, Verification and Reliability, 2009, 19(4): 297－333.
[10] ROTHERMEL G, UNTCH R H, CHU C, et al. Prioritizing Test Cases for Regression Testing [J]. IEEE Transactions on Software Engineering, 2001, 27(10): 929－948.
[11] LI Z, HARMAN M, HIERONS R M. Search Algorithms for Regression Test Case Prioritization [J]. IEEE Transactions on Software Engineering, 2007, 33(4): 225－237.
[12] LIS H, BIAN N W, CHEN Z Y, et al. A Simulation Study on Some Search Algorithms for Regression Test Case Prioritization [C]// 2010 10th International Conference on Quality Software (QSIC). Zhangjiajie, China: IEEE, 2010: 72－81.
[13] KAUR A, GOYAL S. A Genetic Algorithm for Regression Test Case Prioritization Using Code Coverage [J]. International Journal on Computer Science & Engineering, 2011, 3(5): 1839－1847.
[14] 朱海燕，范 辉，谢青松，等. 测试用例排序的研究 [J]. 计算机工程与科学. 2008，30(1)：79－81.
[15] DO H, ELBAUM S, ROTHERMEL G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact [J]. Empirical Software Engineering, 2005, 10(4): 405－435.

Regression Test Case Prioritization Based on AdditionalUL Strategy

TANG Hai-peng[1], DING Xiao-ming[1,2]

1. *School of Computer and Information Science, Southwest University, Chongqing 400715, China;*
2. *Chongqing Software Testing Center Co. Ltd., Chongqing 400715, China*

Abstract: Additional strategy has been widely used in the area of test case prioritization. The sorted results based on it have better error detection rates than other approaches. However, the test cases are ranked randomly if they have the same program coverage based on Additional strategy. Such randomization reduces the effectiveness of the sorted results. With this observation in mind, the authors of this paper propose a new Additional strategy with usage label, abbreviated as AdditionalUL strategy. The new strategy adds tags to the test cases according to their executive information, and then sets the priorities of the test cases. Additional strategy and AdditioanlUL strategy are used to sort the test cases of multiple programs, respectively, and evaluate the sorted results based on APFD. Experimental results show that the strategy proposed herein can improve the effectiveness of the sorted results of Additional strategy and has a higher error finding rate.

Key words: Additional strategy; AdditionalUL strategy; regression testing; test case prioritization; label

责任编辑　崔玉洁