计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(3):217−223 [doi: 10.15888/j.cnki.csa.008960]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

不正确程序修复补丁识别①

董玉坤, 唐道龙, 孙玉雪, 位欣欣

(中国石油大学 (华东) 计算机科学与技术学院, 青岛 266580)
通信作者: 董玉坤, E-mail: dongyk@upc.edu.cn

摘　要: 程序自动修复技术是保证软件质量、提高开发效率的有效手段. 目前, 大多数自动修复工具使用测试用例作为补丁正确性验证的最终方法, 有限的测试用例难以对程序进行充分的测试, 因此自动修复工具生成的补丁集合包含大量的不正确补丁. 为了识别不正确补丁, 我们采用对比缺陷修复前后成功测试的执行路径以及生成测试用例的方法来识别修复补丁的有效性, 以解决自动修复工具精度低的问题. 我们的方法评估了来自 6 个经典的自动修复工具生成的 132 个补丁, 并成功地排除了 80 个不正确的补丁并且没有排除正确的补丁, 这表明我们的方法可以有效地排除不正确补丁, 并且提高自动修复工具的精度.

关键词: 程序自动修复; 补丁质量评估; 不正确补丁; 测试用例生成; 不正确补丁识别; UML; 检测方法

引用格式: 董玉坤,唐道龙,孙玉雪,位欣欣.不正确程序修复补丁识别.计算机系统应用,2023,32(3):217–223. http://www.c-s-a.org.cn/1003-3254/8960.html

Identification of Incorrect Program Repair Patches

DONG Yu-Kun, TANG Dao-Long, SUN Yu-Xue, WEI Xin-Xin

(College of Computer Science and Technology, China University of Petroleum, Qingdao 266580, China)

Abstract: Automatic program repair is an effective technology for ensuring software quality and improving development efficiency. At present, most automatic repair tools use test cases as the final method of patch correctness verification. However, program can barely be fully tested by limited test cases. Consequently, patch sets generated by automatic repair tools contain a large number of incorrect patches. To identify such patches, this study identifies the effectiveness of repair patches by comparing the execution paths of successful tests before and after defect repair and the methods of test case generation to solve the low accuracy problem of automatic repair tools. When the proposed method is applied to evaluate 132 patches generated by six classic repair tools, it successfully excludes 80 incorrect patches, without excluding correct ones. This result shows that the proposed method can effectively exclude incorrect patches and improve the accuracy of automatic repair tools.

Key words: automatic program repair; patch quality evaluation; incorrect patches; test case generation; identification of incorrect patches; UML; detection method

现代软件应用程序的复杂性和规模在不断增加, 导致软件中的缺陷难以检测、定位、修复, 自动化的缺陷检测与程序自动修复得到了越来越多的关注. 特别是近几年, 相关研究人员从不同视角提出了多种程序自动修复方法并研制了相关原型工具[1−6], 目前的程序自动修复方法主要分为两类:生成和验证方法和语义驱动方法.

程序自动修复工具产生的补丁质量决定了该修复

① 基金项目: 山东省自然科学基金 (ZR2021MF058)
收稿时间: 2022-07-26; 修改时间: 2022-08-26; 采用时间: 2022-09-01; csa 在线出版时间: 2022-11-14
CNKI 网络首发时间: 2022-11-16

工具的修复精度, 然而现有的自动修复工具的修复精度很低[7–9], 修复精度低的原因是采用测试用例作为补丁的最终验证方法, 实际工程中的测试用例往往是不充分的[10], 即使通过所有测试, 补丁程序仍然可能存在缺陷. 我们将通过所有测试的补丁称为可疑补丁, 将修复缺陷并不产生副作用的补丁称为正确补丁, 将通过所有测试用例并且没有修复原有缺陷的补丁称为不正确补丁. 根据 Long 等人的研究[11], 大部分可疑补丁是不正确的补丁, Long 等人认为基于测试用例的自动程序修复工具难以确保补丁的正确性. 现有研究[10,12] 表明, 目前绝大多数自动程序修复工具在实际缺陷上产生的错误补丁远多于正确补丁, 导致生成的补丁精度较低. 自动程序修复工具的低准确率问题严重地影响了自动程序修复工具的可用性.

为了识别不正确补丁, 我们采用对比程序修复前后成功测试的执行路径以及生成测试用例的方法来识别不正确补丁, 以解决自动修复工具精度低的问题. 我们的方法可以有效地提高修复工具的修复精度, 并且减少开发者手动排除不正确补丁的时间.

1 自动修复工具产生不正确补丁的原因

自动程序修复工具产生大量不正确补丁的主要原因是测试用例的不充分, 这将导致难以测试程序的所有预期功能, 即弱测试问题. 弱测试问题可以分为两类: 测试的弱断言问题和测试的弱输入问题. 测试的弱断言问题是测试用例仅关注程序是否抛出异常或者仅关注程序的部分预期功能, 而不关心程序完整的预期功能. 测试的弱输入问题是在缺陷路径上, 程序没有充分的测试用例. 在缺陷路径上, 自动工具对已有的少部分测试用例采用不修改路径功能的策略, 对失败的测试用例采用修改路径的功能策略来生成补丁, 这种修改操作可能会影响已有测试用例之外的测试用例的功能. 在测试套件中, 测试用例的弱输入或弱断言问题都可能导致不正确补丁的产生. 为了简化讨论, 我们假设缺陷程序只包含一个缺陷. 为了使补丁通过所有的测试, 自动修复工具修改或删除程序的原始功能或路径, 由于测试的弱断言和弱输入的问题, 自动修复工具没有充分的测试用例来检测修复后的补丁, 所以自动修复工具就产生了不正确补丁.

1.1 测试的弱断言问题

测试的弱断言问题可以使得不正确补丁通过所有测试. 示例 1 展示了一个由自动修复工具 jKali[13] 为在数据集 Defects4j[14] 中的 Chart-15 缺陷生成的不正确的补丁. 在修复缺陷之前, 如果接受的对象是一个空的数据集, 调用 draw 方法将会抛出空指针异常. 示例 2 展示了 Chart-15 缺陷的测试程序, 当接受的对象是一个空的数据集时, 调用修复后 draw 方法不会抛出任何异常, 但是这个补丁跳过整个方法, 这个方法没有执行开发者预期的功能. 弱断言的测试用例仅检查程序有没有抛出异常信息或者程序的部分预期功能, 没有程序的所有预期功能. 同样, 测试的弱输入问题也会导致不正确补丁的生成.

示例1. jKali为Chart-15缺陷生成的不正确补丁

```
public void draw(…){
+ if(true) return ;
…
```

示例2. Chart-15缺陷的测试程序

```
public void testDrawWithNullDataset(){…
  JFreeChart chart = ChartFactory.createPieChart3D("Test", null, …);
  try{
    chart.draw(…);
    success = true;
  }catch(Exception e){
    success = false;
  }
  assertTrue(success);
}
```

1.2 测试的弱输入问题

测试的弱输入问题可以使得不正确补丁非常容易地通过所有测试. 示例 3 展示了一个由自动程序修复工具 Nopol[15] 为在数据集 Defects4j 中的 Lang-39 缺陷生成的不正确的补丁. 当一个数组元素为 null 值时, 缺陷程序将会抛出一个空指针异常. 为了阻止程序抛出空指针异常, 正确方式是跳过那些值为 null 的元素, 如示例 4 所示. 然而, 示例 3 中的补丁基于 *repeat* 的值阻塞了正确的循环语句块, *repeat* 是这个方法的参数. 然而, 这个不正确的补丁能够通过所有测试并且能够产生正确的输出. 原因如下: (1) 对于成功的测试, *repeat* 的值是 true; 对于失败的测试, *repeat* 的值是 false. (2) 失败测试的 *searchList* 和 *replacementList* 的元素不满足条件 greater>0. 由于测试的弱输入问题, 自动修复工具生成的补丁很容易通过所有测试用例, 在缺陷路径上, 仅有少部分测试用例, 没有充分的测试用例对修改后的补丁进行成分测试, 因此产生了不正确补丁.

示例3. Nopol为Lang-39缺陷生成的不正确补丁

```
…
+ if(repeat){
      for(int i=0;i<length;i++){
          int greater = replacementList[i].length()
            -searchList[i].length();
          if(greater>0) increase += 3×greater;
        }
  }
…
```

示例4. Lang-39缺陷的正确补丁

```
…
for(int i=0;i<length;i++){
      + if(searchlist[i] == null ||
      + replacementList[] == null){
      + continue;
      }
      int greater = replacementList[i].length()
          -searchList[i].length();
      if(greater>0) increase += 3×greater;
  }
…
```

2 程序补丁质量评估方法

2.1 识别由仅测试异常的弱断言产生的不正确补丁

缺陷程序所有的测试用例仅关注缺陷程序是否抛出异常, 而且没有测试缺陷程序预期功能的测试用例, 对于这种缺陷程序, 自动修复工具采取跳过整个方法的策略, 就能生成通过所有测试的补丁. 这种补丁删除所有预期的功能, 这种补丁没有任何功能和意义, 这种补丁毫无疑问就是不正确补丁. 因此缺陷程序所有测试用例仅关注是否抛出异常并且修复工具采取跳过缺陷代码的修复策略, 这种补丁就是不正确补丁.

2.2 识别由测试部分功能的弱断言产生的不正确补丁

当缺陷程序的测试用例仅测试程序的部分预期功能并且没有测试缺陷程序全部预期功能时, 自动修复工具为了使得生成的补丁能够通过所有测试用例, 自动修复工具通过修改了缺陷程序的正确功能来生成补丁, 这种修改是不正确的. 因此, 我们提出一种基于路径功能的评估方法, 如果补丁修改了缺陷程序的正确功能, 则这个补丁就是不正确补丁; 如果补丁没有修改缺陷程序的正确功能, 这个补丁是正确的补丁. 现在的问题是如何判断缺陷程序中的每条路径的功能是正确的或者是符合开发者意图的. 在实践中, 我们发现, 当越多的成功测试用例经过的路径时, 它的功能越有可能是正确的. 在现实中, 测试用例的数目是有限的, 因此我们采取了一种折中的方法, 即少数服从多数的原则来判断路径功能的正确性, 成功测试用例数量多于失败测试用例数量的路径是功能正确的路径. 我们使用路径的序列来表示路径的功能. 通过对比成功测试用例在修复缺陷前后的执行路径的差异, 我们可以判断补丁是否修改缺陷程序的正确功能. 如果成功测试在修复前后程序上的执行路径是相同的, 这意味着补丁没有修改缺陷程序的正确功能; 如果成功测试在修复前后程序上的执行路径是不相同的, 这意味着补丁修改了缺陷程序的正确功能; 失败测试用例的执行路径被改变, 并得到正确的输出, 这表明缺陷程序错误路径已经修复, 程序功能恢复[16]. 在原始程序中, 成功测试的行为可以作为程序需要的行为的部分描述[17]. 因此, 我们认为一个合理的补丁不仅能通过所有测试用例, 而且对于成功测试用例, 补丁的执行路径与缺陷程序的执行路径是相同的, 并且失败测试的执行路径是不同的, 则补丁不是正确的补丁, 否则, 补丁就是不正确补丁.

p 表示缺陷程序中的缺陷路径, s 表示通过缺陷路径的成功测试用例的个数, f 表示通过缺陷路径的失败测试用例的个数, $p(i)$ 表示在缺陷程序中通过缺陷路径的第 i ($1\leqslant i\leqslant s$) 个成功测试的执行路径, 当缺陷路径的成功测试用例个数 s 多于失败测试用例个数 f 时, 即 $s>f$, 这表示缺陷路径 p 的功能是正确. $pp(i)$ 表示在补丁中通过缺陷路径的第 i 个成功测试的执行路径. $p(i)==pp(i)$ 表示通过缺陷路径的第 i 个成功测试在缺陷程序中的执行路径与在补丁中的执行路径是相同的, 这表示补丁没有修改缺陷路径的第 i 个测试用例的预期功能, 即 $r(i)=1$. $p(i)!=pp(i)$ 表示通过缺陷路径的第 i 个成功测试在缺陷程序中的执行路径与在补丁中的执行路径存在差异, 这表示补丁修改了通过缺陷路径的第 i 个成功测试的缺陷路径正确功能, 即 $r(i)=0$, 如式 (1) 所示. r 表示补丁是否修改了缺陷路径的正确功能, 如果补丁修改了缺陷程序的正确功能, 即 $r=0$, 则补丁是不正确补丁; 如果补丁没有修改程序的正确功能, 即 $r=1$, 则补丁是正确补丁, 如式 (2) 所示:

$$r(i)=\begin{cases} p(i)==pp(i), & s>f \\ 1, & s\leqslant f \end{cases} \tag{1}$$

$$r=\sum_{i=1}^{s} r(i) \tag{2}$$

示例 5 展示了修复工具 jKali[16] 在数据集 Defects4j

中的 Chart-5 缺陷生成的不正确补丁. 其中, 数字序号表示程序中的要执行语句块, 这些语句块仅包含要实现功能的语句, 不包含控制语句比如 if 语句. 在修复缺陷之前, 当 *index*>0 && *allowDuplicateXValue* == true && *autoSort* == true 时, 数组就会发生索引越界异常. jKali 为了不触发数组越界异常, 直接将表达式 if(*auto-Sort*) 替换为表达式 if(false), 补丁直接跳过了会触发异常的 3 号语句, 去执行 4 号语句. 3 号语句的功能是按照 *x* 值的大小插入元素 (*x*, *y*), 数组中的元素是有序的. 4 号语句的功能是将元素 (*x*, *y*) 追加到数组的末尾, 数组中的元素是无序的. 这个补丁破坏了程序预期的排序功能.

示例5. jKali生成的不正确补丁

```
…                                                          1
if(index>=0&&! allowDuplicateXValues){
    …                                                      2
}else{
    -if(autoSort){
    +if (false) {
        this.data.add(-index-1, x, y);                     3
    }else{
        this.data.add(x, y);                               4
    }
…                                                          5
```

在表 1 中展示了{1, 3, 5}缺陷程路径 *p* 的 3 个成功测试和一个失败测试, 在缺陷路径{1, 3, 5}上成功测试的数量 *s* 多于失败测试的数量 *f*, 这表示缺陷路径 *p* 的功能是正确的. 1, 2 和 3 号测试都是成功的测试, 4 号测试是失败的测试. 1, 2 和 3 号成功测试在缺陷程序中的执行路径都是{1, 3, 5}. 1, 2 和 3 号成功测试在补丁中的执行路径都是{1, 4, 5}. 1, 2 和 3 号成功测试在缺陷程序和补丁中的执行路径发生了改变, 缺陷路径 *p* 的正确功能发生了改变. 所以, *pp*(*i*)==*p*(*i*) 的结果如表 1 所示, 最后, *r*=0 表示补丁是不正确补丁.

$$r = \prod_{i=1}^{3} r(i) = 0$$

表 1 Chart-5 缺陷路径上已存在的测试

序号*i*	测试	*pp*(*i*)	*p*(*i*)	执行结果	*p*(*i*)==*pp*(*i*)
1	(1, 2)	1, 3, 5	1, 4, 5	成功执行	0
2	(2, 3)	1, 3, 5	1, 4, 5	成功执行	0
3	(1, 1)	1, 3, 5	1, 4, 5	成功执行	0
4	(1, 4)	1, 3	1, 4, 5	抛出异常	—

2.3 识别由弱输入造成的不正确补丁

弱输入的用例本身不能够识别出测试弱输入问题造成的不正确补丁, 为了能够识别由弱测试输入问题产生的不正确补丁, 直观的想法是增强测试用例. 有很多研究[18,19] 已经尝试生成新的测试用例来识别不正确的补丁. 然而, 测试输入容易地生成, 测试预言不能自动地生成[20]. Xiong 等人[21] 发现当两个测试用例有相同的执行路径时, 他们很有可能有相同的测试预言, 这两个测试用例要么都触发相同的错误, 要么都正常执行, 我们把这种现象称之为测试用例相似性. 我们利用测试用例相似性来解决生成测试预言的问题, 我们假设: 当新生成的测试输入的执行路径与失败测试用例的执行路径相同, 新生成的测试输入会产生不正确的输出; 当新生成的测试输入的执行路径与正确测试用例的执行路径相同, 新生成的测试输入会产生正确的输出. 根据上面的假设, 通过新生成测试输入的执行路径和已经存在的测试用例的执行路径, 我们可以将新生成的测试输入分类为成功的测试输入和失败的测试输入.

g 表示新生成的测试输入, *t* 表示已有的测试用例输入, *X*(*g*) 表示新生成的测试输入 *g* 的在缺陷程序的执行路径, *Y*(*t*) 表示已存在的测试用例 *t* 的执行路径, 我们使用最大公共子序列 (*LCS*) 来计算新生成测试输入的执行路径 *X*(*g*) 和已有测试输入的执行路径 *Y*(*t*) 之间的距离, 最后根据新生成测试输入与已有测试用例之间距离, 将新生成的测试输入分为成功的测试输入或者失败的测试输入, 距离计算公式为式 (3). *Oracle*(*g*) 表示新生成测试输入 *g* 的分类结果, pass 表示测试输入 *g* 为成功的测试输入, fail 表示测试输入 *g* 是失败的测试输入, discard 表示丢弃新生成的测试输入 *g*, 新生成测试输入的分类公式如式 (4) 所示:

$$Dist(X(g), Y(t)) = \frac{|LCS(X(g), Y(t))|}{\max(|X(g)|, |Y(t)|)} \tag{3}$$

$$Oracle(g) = \begin{cases} \text{pass}, & Dp = 1 \\ \text{fail}, & Df = 1 \\ \text{discard}, & \text{others} \end{cases} \tag{4}$$

其中, $Dp = \max(\{Dist(g, t) | t \text{ 是成功的测试}\})$, $Df = \max(\{Dist(g, t) | t \text{ 是失败的测试}\})$. 注意: 已有的测试用例中必须包含成功测试用例, 否则, 将无法将新生成的测试输入分类为成功的测试和失败的测试.

对于被分类为成功的测试输入, 我们利用缺陷程序为成功测试输入生成测试预言, 将测试输入和测试预言作为补丁的测试用例, 补丁无法通过新增的测试用例, 我们认为这是不正确补丁. 对于分类为失败的测

试输入, 收集失败的测试输入在缺陷程序中的执行路径和在补丁程序中的路径, 我们对比失败测试用例在缺陷程序和补丁的执行路径的差异, 如果两者执行路径相同, 说明补丁程序没有完全修复缺陷程序, 这说明这个补丁是不正确; 如果路径不同, 我们无法给出判断.

3 实验分析

3.1 数据集

我们选择 6 个经典修复工具生成的补丁作为数据集, 表 2 展示了数据集的统计结果. 数据集包含 6 个修复工具生成的补丁. 这些工具中, jGenProg[13] 是 GenProg[8] 的 Java 实现版本, 是一个基于遗传规划算法的修复工具; jKali 是 Kali 的 Java 的实现版本, 这是一个仅删除功能的修复工具; Nopol[15] 是一个依赖于约束求解来修复不正确条件的修复工具, 分为两个版本 Nopol2015[22] 和 Nopol2017[15]; DynaMoth[23] 是一种扩展 Nopol 的修复工具, 它可以求解 Nopol 无法合成条件的问题, 比如方法调用; ACS[24] 是一种利用多种数据源的统计信息来修复不正确的条件. 我们选择的工具覆盖了 3 种类型的补丁生成方法: 搜索算法 (jGenProg, jKali)、约束求解算法 (Nopol2015, Nopol2017, DynaMoth) 和基于统计信息的算法 (ACS), 分析质量评估方法在不同类型的自动修复工具上的效果, 可以更全面地衡量补丁质量评估方法的评估能力.

表 2 6 个修复工具的补丁统计

工程	jGenProg			jKali			Nopol2015			Nopol2017			DynaMoth			ACS			Total		
	P	C	O	P	C	O	P	C	O	P	C	O	P	C	O	P	C	O	P	C	O
Chart	6	0	6	6	0	6	6	1	5	6	0	6	8	0	8	2	2	0	34	3	31
Lang	0	0	0	0	0	0	7	3	4	4	0	4	1	0	1	3	1	2	15	4	11
Math	14	5	9	10	1	9	15	1	14	22	0	22	16	0	16	15	11	4	92	18	74
Time	2	0	2	2	0	2	1	0	1	8	0	8	3	0	3	1	1	0	17	1	16
Total	22	5	17	18	1	17	29	5	24	40	0	40	28	0	28	21	15	6	158	26	132

注: P表示可疑补丁的数量, C表示正确补丁的数量, O表示不正确补丁的数量.

3.2 实验设置

对于弱测试断言问题, 我们通过补丁和缺陷修复报告获取缺陷程序的缺陷路径, 在缺陷路径中插入打印调用栈的代码, 将所有调用修改后的方法的测试用例和调用栈信息输出到指定文件中, 在 Defects4j 中, 我们使用编译和测试命令, 对修改后的程序进行编译和测试, 我们可以获得经过缺陷路径的所有测试用例和调用栈信息, 这些测试用例包括失败的测试用例和成功的测试用例. 我们使用符号执行技术[25] 可以获取测试用例在缺陷程序和补丁中的执行路径. 对于弱测试输入问题, 我们需要生成测试输入来覆盖修改后的方法, 我们使用测试输入生成工具 Randoop[26] 来自动地生成测试输入, 我们将 Randoop 生成测试输入的运行时间设置为 30 min, 并且利用符号执行技术来筛选出覆盖修改后路径的测试输入. 然后根据上文中提出的测试用例分类方法, 将生成的测试用例分类为成功测试和失败测试. 对于被分类成功的测试输入, 直接输入到缺陷程序, 可以得到正确的测试预言, 测试输入和测试预言可以生成增强的测试用例.

3.3 补丁评估结果

表 3 展示了我们方法不同工具在数据集上的效果, 如表 3 所示, 我们的方法从 132 个可疑补丁中排除掉 80 个不正确的补丁, 并且没有排除正确补丁. 此外, 我们的方法在不同工具表现出相似的性能, 这表示我们的结果能够推广到不同类型的项目和不同类型的工具. 从理论上讲, 我们的方法是有可能排除正确补丁.

表 3 多种工具补丁的评估结果

工具	不正确补丁	正确补丁	排除的不正确补丁	排除的正确补丁
jGenProg	17	5	13 (76.5%)	0
jKali	17	1	10 (58.8%)	0
Nopol2015	24	5	16 (66.7%)	0
Nopol2017	40	0	20 (50.0%)	0
ACS	6	15	4 (66.7%)	0
DynaMoth	28	0	17 (60.7%)	0
总计	132	26	80 (43.2%)	0

3.4 其他评估方法对比

为了说明我们方法的有效性, 我们的方法与另外 4 种程序补丁评估方法进行对比. 反模式[27] 能从 139 个补丁中排除 28 个补丁, 包含 27 个不正确补丁和 1 个正确补丁, 结果显示我们的方法要优于反模式. 生成测试用例方法[21] 能从 110 个不正确的补丁中排除 62 个不正确补丁并且没有排除正确补丁, 我们的方法的效果优于生成测试用例的方法. 基于语法距离的方

法[28,29] 可以排除 56.3% 的不正确补丁同时能排除 66.7% 的正确补丁, 结果说明语法距离不能直接用来判断补丁的正确性. 基于语义距离的方法[28,30] 可以排除 56.3% 的不正确补丁并且能排除 44.3% 的正确补丁, 结果说明语义距离不能直接用来判断补丁的正确性.

4 结论与展望

在本文中, 我们采用对比程序修复前后成功测试的执行路径以及生成测试用例的方法来识别不正确补丁的方法, 正如实验结果所示, 我们的方法可以有效地过滤掉 43.2% 的不正确补丁, 因此提高自动修复工具的修复准确率, 并且没有过滤掉任何正确的补丁. 结果表示通过对比修改前后成功测试的执行路径和生成测试用例是一个很有效的方法. 我们希望能找到更多的方法来识别不正确补丁, 解决自动修复工具精度率低的问题.

参考文献

1 Ming W, Chen JJ, Wu RX, *et al*. Context-aware patch generation for better automated program repair. Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering. Gothenburg: IEEE, 2018. 1–11. [doi: 10.1145/3180155.3180233]

2 Jiang JJ, Xiong YF, Zhang HY, *et al*. Shaping program repair space with existing patches and similar code. Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. Amsterdam: ACM, 2018. 298–309. [doi: 10.1145/3213846.3213871]

3 Nguyen HDT, Qi DW, Roychoudhury A, *et al*. SemFix: Program repair via semantic analysis. Proceedings of the 2013 35th International Conference on Software Engineering. San Francisco: IEEE, 2013. 772–781. [doi: 10.1109/ICSE.2013.6606623]

4 Yuan Y, Banzhaf W. ARJA: Automated repair of Java programs via multi-objective genetic programming. IEEE Transactions on Software Engineering, 2018, 46(10): 1040–1067.

5 Saha RK, Lyu Y, Yoshida H, *et al*. Elixir: Effective object-oriented program repair. Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. Urbana: IEEE, 2017. 648–659. [doi: 10.1109/ASE.2017.8115675]

6 Xin Q, Reiss SP. Leveraging syntax-related code for automated program repair. Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. Urbana: IEEE, 2017. 660–670. [doi: 10.1109/ASE.2017.8115676]

7 Gazzola L, Micucci D, Mariani L. Automatic software repair: A survey. IEEE Transactions on Software Engineering, 2019, 45(1): 34–67. [doi: 10.1109/TSE.2017.2755013]

8 Le Goues C, Nguyen TV, Forrest S, *et al*. GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering, 2012, 38(1): 54–72. [doi: 10.1109/TSE.2011.104]

9 Le Goues C, Dewey-Vogt M, Forrest S, *et al*. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. Proceedings of the 2012 34th International Conference on Software Engineering. Zurich: IEEE, 2012. 3–13. [doi: 10.1109/ICSE.2012.6227211]

10 Qi ZC, Long F, Achour S, *et al*. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. Proceedings of the 2015 International Symposium on Software Testing and Analysis. Baltimore: ACM, 2015. 24–36. [doi: 10.1145/2771783.2771791]

11 Long F, Rinard C. An analysis of the search spaces for generate and validate patch generation systems. Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering. Austin: IEEE, 2016. 702–713. [doi: 10.1145/2884781.2884872]

12 Smith EK, Barr ET, Le Goues C, *et al*. Is the cure worse than the disease? Overfitting in automated program repair. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 532–543. [doi: 10.1145/2786805.2786825]

13 Martinez M, Monperrus M. ASTOR: A program repair library for Java (demo). Proceedings of the 25th International Symposium on Software Testing and Analysis. Saarbrücken: ACM, 2016. 441–444. [doi: 10.1145/2931037.2948705]

14 Dong YK, Zhang L, Pang SC, *et al*. Automatic repair of semantic defects using restraint mechanisms. Symmetry, 2020, 12(9): 1563. [doi: 10.3390/sym12091563]

15 Xuan JF, Martinez M, DeMarco F, *et al*. Nopol: Automatic repair of conditional statement bugs in Java programs. IEEE Transactions on Software Engineering, 2017, 43(1): 34–55. [doi: 10.1109/TSE.2016.2560811]

16 Dong YK, Wu M, Zhang L, *et al*. Priority measurement of patches for program repair based on semantic distance. Symmetry, 2020, 12(12): 2102. [doi: 10.3390/sym12122102]

17 Tao YD, Kim J, Kim S, *et al*. Automatically generated patches as debugging aids: A human study. Proceedings of

the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Hong Kong: ACM, 2014. 64–74. [doi: 10.1145/2635868.2635873]

18 Yang JQ, Zhikhartsev A, Liu YF, *et al*. Better test cases for better automated program repair. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 831–841. [doi: 10.1145/3106237.3106274]

19 Qi X, Reiss SP. Identifying test-suite-overfitted patches through test case generation. Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. Santa Barbara: ACM, 2017. 226–236. [doi: 10.1145/3092703.3092718]

20 Barr ET, Harman M, McMinn P, *et al*. The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering, 2015, 41(5): 507–525. [doi: 10.1109/TSE.2014.2372785]

21 Xiong YF, Liu XY, Zeng MH, *et al*. Identifying patch correctness in test-based program repair. Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering. Gothenburg: IEEE, 2018. 789–799. [doi: 10.1145/3180155.3180182]

22 Martinez M, Durieux T, Sommerard R, *et al*. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4j dataset. Empirical Software Engineering, 2017, 22(4): 1936–1964. [doi: 10.1007/s10664-016-9470-4]

23 Durieux T, Monperrus M. DynaMoth: Dynamic code synthesis for automatic program repair. Proceedings of the 2016 IEEE/ACM 11th International Workshop in Automation of Software Test. Austin: IEEE, 2016. 85–91. [doi: 10.1145/2896921.2896931]

24 Xiong YF, Wang J, Yan RF, *et al*. Precise condition synthesis for program repair. Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering. Buenos Aires: IEEE, 2017. 416–426. [doi: 10.1109/ICSE.2017.45]

25 赵跃华, 阚俊杰. 基于符号执行的测试数据生成方法的研究与设计. 计算机应用与软件, 2014, 31(2): 303–306. [doi: 10.3969/j.issn.1000-386x.2014.02.081]

26 Pacheco C, Ernst MD. Randoop: Feedback-directed random testing for Java. Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion. Montreal: ACM, 2007. 815–816. [doi: 10.1145/1297846.1297902]

27 Tan SH, Yoshida H, Prasad MR, *et al*. Anti-patterns in search-based program repair. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Seattle: ACM, 2016. 727–738. [doi: 10.1145/2950290.2950295]

28 Le XBD, Chu DH, Lo D, *et al*. S3: Syntax-and semantic-guided repair synthesis via programming by examples. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 593–604. [doi: 10.1145/3106237.3106309]

29 Mechtaev S, Yi J, Roychoudhury A. DirectFix: Looking for simple program repairs. Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Florence: IEEE, 2015. 448–458. [doi: 10.1109/ICSE.2015.63]

30 D'Antoni L, Samanta R, Singh R. Qlose: Program repair with quantitative objectives. Proceedings of the 28th International Conference on Computer Aided Verification. Toronto: Springer, 2016. 383–401. [doi: 10.1007/978-3-319-41540-6_21]

(校对责编: 孙君艳)