计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications [doi: 10.15888/j.cnki.csa.009312]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于数据流分析的过拟合补丁识别①

董玉坤, 杨宇飞, 程小彤, 唐叶尔

(中国石油大学 (华东) 青岛软件学院、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: 自动程序修复技术可实现对软件缺陷的自动修复, 并使用测试套件评估修复补丁. 然而因为测试套件不充分, 通过测试套件的补丁可能并未正确修复缺陷, 甚至引入新的缺陷并产生波及效应, 导致自动程序修复生成大量过拟合补丁. 针对这个问题, 本文提出了一种基于数据流分析的过拟合补丁识别方法, 首先将补丁对程序的修改分解为对变量的操作, 然后采用数据流分析方法识别补丁影响域, 并根据补丁影响域选择针对性覆盖准则来识别目标覆盖元素, 进而选取测试路径并生成测试用例实现对修复程序的充分测试, 避免修复副作用的影响. 本文在两个数据集上进行了评估, 实验结果表明, 基于数据流分析的过拟合补丁识别方法可有效提升自动程序修复的正确性.
关键词: 自动程序修复; 过拟合补丁; 补丁影响域; 数据流分析; 测试用例生成

引用格式: 董玉坤,杨宇飞,程小彤,唐叶尔.基于数据流分析的过拟合补丁识别.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/9312.html

Overfitting Patch Identification Based on Data Flow Analysis

DONG Yu-Kun, YANG Yu-Fei, CHENG Xiao-Tong, TANG Ye-Er

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

Abstract: Automatic program repair techniques can realize automatic repair of software defects and employ test suites to evaluate repair patches. However, because of inadequate test suites, the patches passing the test suites may not repair the defects correctly, or even introduce new defects with ripple effects, which results in a large number of overfitting patches generated by automatic program repair. To this end, an overfitting patch identification method based on data flow analysis is proposed. This method firstly decomposes the patch modifications to the program into operations on variables, then adopts data flow analysis to identify the patch influence domain, and selects targeted coverage criteria to identify target coverage elements according to the domain. Finally, test paths are selected and test cases are generated to fully test the repair program to avoid the impact of repairing side effects. This study conducts evaluations on two datasets, and the experimental results show that the overfitting patch identification method based on data flow analysis can improve the correctness of automatic program repair.
Key words: automatic program repair (APR); overfitting patches; patch influence domain; data flow analysis; test case generation

程序缺陷是软件开发及维护工作中难以避免的问题, 可引起软件故障甚至是系统崩溃. 而修复缺陷是软件开发中最耗费人力的任务之一. 因此人们致力于研究自动程序修复 (automatic program repair, APR), 旨在自动生成补丁来减少手动修复错误的工作量[1].

近几年, 业界提出了许多自动程序修复技术[2–6]. 这些技术都将测试套件作为补丁验证的标准, 旨在生成一个通过所有测试用例的补丁. 然而实践中的测试套件往往不充分, 通过所有测试用例的补丁可能仍然没有修复缺陷, 甚至引入新的缺陷并产生波及效应. 因

① 基金项目: 山东省自然科学基金 (ZR2021MF058)
收稿时间: 2023-05-18; 修改时间: 2023-06-26; 采用时间: 2023-07-03; csa 在线出版时间: 2023-09-21

此, 人们将通过测试套件的补丁表示为合理的补丁. 同时, 若该补丁正确修复错误且未对程序产生副作用,则为正确的补丁, 否则为过拟合补丁. 如 Long 等人[7] 研究所示, 现有的程序修复系统生成的合理的补丁大部分是过拟合补丁, 一项现有研究[8] 还表明, 当开发人员获得不正确补丁时, 修复中产生的波及效应会降低软件的性能, 从而导致用户对修复结果不认可, 制约着 APR 工具的推广应用.

我们的目标是分析补丁修改的波及效应, 避免修复副作用的影响, 从而识别更多的过拟合补丁. 目前 APR 工具采用变异测试的思想, 对程序语句、表达式进行细粒度的修改, 导致程序状态与数据流信息发生改变, 并因为波及效应导致程序产生多粒度成分的变更. 因此, 我们将程序的修改分解为对变量的操作 (添加、修改、删除), 基于数据流分析方法为不同变量 (局部变量、常量、全局变量) 制定相应的分析机制, 充分利用变量间的关联关系, 结合过程内分析与过程间分析, 确定补丁程序中与修改变量有关联关系的节点, 将其加入补丁影响域中, 作为分析结果的集合. 然后根据补丁影响域选择针对性覆盖准则来识别目标覆盖元素, 最后选取路径并生成测试用例实现对程序的充分测试. 我们的方法能有效提高程序修复系统的精度, 降低开发人员手动验证补丁的代价.

本文第 1 节详细介绍补丁影响域的识别过程. 第 2 节解释如何匹配针对性覆盖准则以提取目标覆盖元素. 第 3 节阐述如何通过覆盖元素选取测试路径并生成测试用例. 第 4 节展开了一系列的实验对比讨论, 评估本文方法的有效性. 第 5 节对本文工作进行总结, 同时展望下一步的研究工作.

1 补丁影响域识别

补丁中的代码修改是识别补丁影响域的前提. 虽然补丁程序中语句、表达式的修改相对复杂, 但它们包含的变量类型 (局部变量、常量、全局变量) 及其更改类型 (添加、修改、删除) 是有限的, 因此我们只需为这些有限类型的变量制定分析机制, 最终的修改影响是它们分析结果的集合. 语句、表达式中包含的变量及其更改类型如表 1 所示.

表 1 变量及更改类型

更改类型	解释	更改类型	解释
ALVD	增加局部变量Add a local variable (a variable declaration)	DCV	删除常量 Delete a const variable
CLVV	修改局部变量Change a local variable value	AGVD	增加全局变量Add a global variable (a variable declaration)
CLVL	修改变量上下限Change a local variable upper/lower limits	CGVV	修改全局变量Change a global variable value
DLV	删除局部变量 Delete a local variable	DGV	删除全局变量 Delete a global variable
ACVD	增加常量Add a const variable (a variable declaration)	CPM	修改方法参数Change a parameter of the method
CCVV	修改常量Change a const variable value	CRM	修改方法返回值Change a return value of the method

下面显示了 3 个补丁修改示例, 包括对条件语句、表达式操作符以及赋值语句的修改. 如示例 3, 工具修改了变量 *index* 的赋值表达式. 因此在影响分析中, 我们首先将该补丁修改分解为对 *index* 变量的操作, 基于数据流分析方法识别变量 *index* 的修改影响.

示例 1. Nopol 修复 FirstZero-bug9 生成的补丁

```
– if (x.length == 0)
+ if (x.length == 1)
```

示例 2. jMutRepair 修复 Bank-bug6 生成的补丁

```
– int mid = start + (end – start)/2
+ int mid = start + (end + start)/2
```

示例 3. Cardumen 修复 Inverse-bug2 生成的补丁

```
– index = index + 1
+ index = x[index]
```

此外, 有些修改产生方法内影响的同时还会产生跨方法的影响. 因此生成 CFG 时还要分析程序的调用关系, 从而进行过程间分析. 示例 4 是工具 ArjaE 生成的一个修复补丁, 该修复产生了跨方法的影响.

示例 4. Combination-bug3 修复补丁

```
1. public class CombinationPermutation {
2.   public long combination(int n, int r) {
3.       Factorial fac = new Factorial();
4.       long combin;
5.       – combin = fac.factorial(n)*(fac.factorial(r)*fac.factorial(n–r));
         + combin = fac.factorial(n)/(fac.factorial(r)*fac.factorial(n–r));
6.   return combin;
7.   }
8.   public long permutation(int n, int r) {
9–12. …
13.  }
14.     public long select(int n, int r, boolean flag) {
15.  return flag ? combination(n, r) : permutation(n, r);
```

```
16.   }
17. }
```

图 1 是该程序带有调用关系的 CFG. 其中, V_1 节点为 ArjaE 的修改节点, V_2, V_3, V_5 节点为受修复影响的节点, 两种节点共同构成 Combination-bug3 的补丁影响域. 接下来将详细介绍补丁影响域的识别过程.

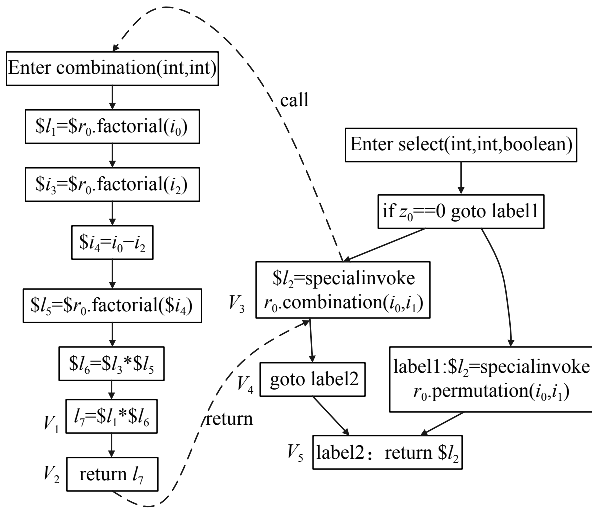

图 1 Combination-bug3 补丁影响域

1.1 过程内影响域识别

过程内影响域识别采用数据流迭代算法[9], 基于控制流图 (control flow graph, CFG)[10] 进行流敏感的分析. 对 CFG 上的每个节点 n, 数据流计算方程如下:

$$in(n)=\begin{cases} init(n), & n=\text{entry} \\ \bigcup_{p\in pred(n)} out(p), & \text{otherwise} \end{cases} \tag{1}$$

$$out(n)=gen(n)\cup(in(n)-kill(n)) \tag{2}$$

其中, n 为控制流图当前节点, p 为 n 前驱节点. 式 (1) 中 $in(n)$ 计算 n 的汇入信息; $init(n)$ 初始化方法入口处的全局变量与参数; 式 (2) 中 $out(n)$ 计算 n 的出口信息; $gen(n)$ 表示 n 中新产生的数据流信息; $kill(n)$ 表示 n 中注销或改变的数据流信息; $pred(n)$ 表示 n 的所有前驱节点集合.

基于计算的数据流信息, 我们将根据算法 1 查找过程内路径来获取与修改变量相关的数据流状态发生改变的节点. 第 1 步获取该点的所有出边并存储在 etable 中; 第 2 步对 etable 进行迭代, 每次获取一条边及该边的 endNode. 如果该 endNode 未被访问, 则添加到节点列表. 如果所有的 endNode 都已加入路径集或者在第 1 步中该节点不存在出边, 则将此类节点当作程序终点, 并将它所在的节点序列添加至路径集.

算法 1. 查找过程内分析路径算法

输入: 控制流图 CFG, 图上节点 v, 节点列表 NList.
输出: 一组用于过程内分析的路径 paths.

说明: getOutedges() 为获取一个节点的所有出边
getEndNode() 为获取一条边的尾节点

```
BEGIN
1. Hashtable<String, Edge>etable=v.getOutedges()
2. if (etable is empty) then
3.    paths.add(NList);
4. else
5.    foreach item in etable do
6.       edge←etable.get(item);
7.       endNode←edge.getEndNode();
8.       if (endNode not in NList) then:
9.          NList.add(endNode);
10.         findPath(NList, endNode, paths);
11.         newNode– –;
12.   if (newNode == 0) then
13.      paths.add(NList);
END
```

基于过程内路径, 我们将变量的修改点 (change point, cp) 分为定义点和使用点, 并使用影响域 (influence domain, ID) 存储与修改点有关联关系的节点, 由于修改影响会迭代产生, 因此我们基于以下公式获取这些节点.

$$ID_0=F(cp) \tag{3}$$

$$ID_1=F(ID_0-cp) \tag{4}$$

$$ID_n=F(ID_{n-1}-\bigcup_{i=0}^{n-2} ID_i)\ (n\geqslant 2) \tag{5}$$

$$ID=\bigcup_{i=0}^{n} ID_i \tag{6}$$

接下来, 我们将详细介绍识别补丁影响域的方法, 即上述公式中的 F. 我们将影响域 ID 分为两类: ID_{define} 和 ID_{use}, 其中, $ID_{\text{use}}=\{ID_{\text{c-use}}, ID_{\text{p-use}}\}$. $ID_{\text{c-use}}$ 中的元素对修改点进行计算使用, $ID_{\text{p-use}}$ 中的元素对修改点进行谓词使用. 算法 2 显示了根据不同类型的修改点获得分类的补丁影响域的过程.

算法 2. 识别补丁影响域算法

输入: 控制流图 CFG, 修改点 cp, 过程内路径 paths.
输出: 补丁影响域 ID_{type}, type={define, c-use, p-use}.

说明: getCanculateUse() 为获取计算使用 cp 变量的节点
getPredicateUse() 为获取谓词使用 cp 变量的节点

```
BEGIN
1. if (cp.type is use) then
```

```
2.     vex←getCanculateUse();
3.     ID_c-use.add(vex);
4.     vex←getPredicateUse();
5.     ID_p-use.add(vex);
6. else if (cp.type is definition) then
7.     ID_define.add(cp);
8.     foreach path in paths do
9.        if (∃vex in path uses cp) then
10.          vex←getCanculateUse();
11.          ID_c-use.add(vex);
12.          vex←getPredicateUse();
13.          ID_p-use.add(vex);
END
```

1.2 过程间影响域识别

过程间影响域识别的目标是识别补丁修改的跨方法影响, 因此它分析的目标点 (target point, tp) 是变更的方法参数、返回值和全局变量所在的节点, 分析的结果是程序中受影响的方法集. 此时目标点有 3 种类型: {call, return, global}. 其中, call 表示目标点上方法参数更改, return 表示目标点上返回值更改, global 表示目标点上全局变量更改. 分析结束后将继续进行过程内影响域识别.

我们根据算法 3 查找受影响的方法集. 对于方法返回值的变更, 首先根据方法之间的关系找到调用它的方法, 然后将该方法添加到过程间影响域 IMD 中; 当分析方法调用时, 我们需要获取它调用的方法或者根据 SEA 关系[11] 获得受影响的方法, 并将其添加到 IMD 中; 对于全局变量, 使用数据流分析确定它的使用点, 这些使用点所在的新方法也将添加到 IMD 中.

算法 3. 识别过程间影响域算法

输入: 控制流图 CFG, 目标点 tp.
输出: 过程间影响域 IMD (受影响的方法集).

```
说明: getReturn(tp) 为获取使用 tp 返回值的方法
getCall(tp) 为获取 tp 调用的方法
getExecute(tp) 获取与 tp 具有 SEA 关系的方法
getGlobal(tp) 为获取使用 tp 全局变量的方法
BEGIN
1. if (tp.type==“return”) then
2.     new_method←getReturn(tp);
3.     IMD.add (new_method);
4. else if (tp.type==“call”) then
5.     new_method←getCall(tp) or getExecute(tp);
6.     IMD.add (new_method);
7. else if (tp.type==“global”) then:
8.     new_method←getGlobal(tp);
9.     IMD.add (new_method);
END
```

2 覆盖元素提取

覆盖元素提取是基于覆盖准则实现的, 目的是确定测试用例必须覆盖的程序元素. 上面我们提到, 补丁影响域识别是基于数据流分析方法通过分析变量的定义与使用来实现的. 因此我们主要关注与修改变量相关的 def-use 对.

我们在数据流分析中区分了两种类型的变量使用: 计算使用 (即 c-use), def-use 对表示为 dcu(V_d, V_u, x); 谓词使用 (即 p-use). 此时, 出现两个 def-use 对, 分别表示为 dpu(V_d, (V_u, V_t), x) 和 dpu(V_d, (V_u, V_f), x), 其中 x 在 V_u 处有两个不同的分支 (V_u, V_t) 和 (V_u, V_f): 前者表示使用 x 的条件语句的真分支, 后者是假分支. 因此, 覆盖元素提取将基于 c-use 覆盖、p-use 覆盖提取目标覆盖元素, 增强程序测试的充分性.

3 测试路径选取

提取覆盖元素之后, 需要选择一条或多条包含覆盖元素的测试路径. 基于待覆盖节点, 根据补丁程序中节点的控制关系和蕴含关系找出从控制流图入口到目标节点以及从该节点到控制流图出口的必经节点, 基于这些节点选取测试路径. 假设 V_a, V_b 为必经节点序列中的两个节点, 我们首先从节点 V_a 的子节点中找出一个能到达 V_b 的节点 V_c, 然后填充 V_c 到 V_b 之间的路径. 由此可见, 节点 V_c 的选取是填充 V_a 和 V_b 间路径的关键. 因此, 我们基于以下原则选择节点 V_c.

(1) 若节点 V_a, V_b 间不存在覆盖元素, 则遵循最短路径选择.

(2) 若存在待覆盖节点 V_g, 且该节点可到达 V_a 的另一个子节点, 则选取 V_g 作为节点 V_c.

(3) 若节点 V_a 不存在满足 (2) 的子节点, 则选取 V_a 的一个待覆盖子节点作为节点 V_c.

(4) 若 V_c 为循环判定节点, 则该节点最多在路径上出现 2 次, 否则为 1 次 (该原则可有效解决实际执行时的无限循环问题).

此外, 将选取的路径用于生成测试用例之前还需进行不可达路径检测, 以降低测试用例生成的代价. 我们基于数据流分析, 通过区间运算计算路径上的变量和表达式的取值范围, 从而确定待测路径是否为不可达路径. 如果路径 path 的一个节点上变量 a 的取值为空, 则说明该节点与之前某节点矛盾, 从而判定 path 为不可达路径.

4 实验分析

4.1 数据集

我们基于两个数据集 QuixBugs[12] 和 BuggyJava-JML[13], 包括 jGenprog[6], RSRepair[14], Nopol[3], DynaMoth[15], JMutRepair[16], SimFix[2] 生成的 165 个补丁上评估了我们的方法. 表 2、表 3 展示了数据集的评估结果. 其中, jGenprog 和 RSRepair 采用遗传算法, 通过定义代码片段的交叉和变异操作生成修复补丁; Nopol 将程序中的变量用于生成补丁的同时还编码了一些常量 (如 0、1 等) 来支持补丁生成; DynaMoth 针对方法调用进行约束求解来修复程序; JMutRepair 通过改变条件语句中的表达式来生成补丁; SimFix 替换相似语句中的变量来生成补丁. 这些工具覆盖了两种通用性较强的补丁生成方法: 基于启发式搜索 (jGenprog, RSRepair, SimFix, JMutRepair)[17] 和基于语义约束 (Nopol, DynaMoth). 并且由于大多数动态 APR 工具只能修复带有一个缺陷的程序, 因此这两个数据集中每个程序都只有一个缺陷, 且每个程序都最大程度地保留了原始程序的特性. 与 $GT_{Invariants}$[18]、$RGT_{InputSampling}$[19] 以及 OpenJML[13] 在上述两个数据集上对比识别过拟合补丁的效果, 同时计算每种技术的准确率, 以充分例证本文方法的有效性.

表 2 QuixBugs 数据集实验结果对比

方法	问题	DP	QI	LI	VA	KN	LE	ME	PE	PO	RP	FA	SO	DT	SQ	总数	准确率
	补丁	24	11	9	1	1	1	1	1	1	1	1	1	1	18		
Our method	过拟合	23	0	0	1	0	0	0	1	1	0	1	1	0	18	46	0.972
	误报	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	正确	0	11	9	0	1	1	1	0	0	1	0	0	0	0	24	
	漏报	1	0	0	0	0	0	0	0	0	0	0	0	1	0	2	
Invariants	过拟合	20	0	0	1	0	0	0	0	0	0	0	1	1	18	41	0.736
	误报	0	3	9	0	0	0	0	0	0	0	0	0	0	0	12	
	正确	0	8	0	0	1	1	1	0	0	1	0	0	0	0	12	
	漏报	4	0	0	0	0	0	0	1	1	0	1	0	0	0	7	
InputSampling	过拟合	0	0	0	1	0	0	0	0	1	0	0	1	0	18	21	0.639
	误报	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	正确	0	11	9	0	1	1	1	0	0	1	0	0	1	0	25	
	漏报	24	0	0	0	0	0	0	1	0	0	1	0	0	0	26	
Manual	—	24	0	0	1	0	0	0	1	1	0	1	1	1	18	48	—

表 3 BuggyJML 数据集实验结果对比

方法	程序	AD	CA	PE	SO	FI	GC	IN	LI	OD	CH	SM	FIB	NU	BIN	FA	IN	BA	AB	ST	TI	STU		
	补丁	6	2	3	3	3	1	5	2	1	2	5	3	5	12	3	6	4	1	5	7	14	总数	准确率
Our method	过拟合	0	0	0	1	1	0	4	0	0	1	3	0	1	1	0	1	3	1	5	5	12	39	0.946
	误报	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	正确	6	2	3	2	1	1	1	2	1	1	2	3	4	11	3	4	0	0	0	0	2	49	
	漏报	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	2	0	5	
OpenJML	过拟合	0	0	0	1	2	0	4	0	0	1	3	0	1	1	0	2	4	1	5	7	12	44	0.710
	误报	4	1	1	1	0	0	0	1	0	0	1	2	3	6	2	3	0	0	0	0	2	27	
	正确	2	1	2	1	1	1	1	1	1	1	1	1	1	5	1	1	0	0	0	0	0	22	
	漏报	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
InputSampling	过拟合	0	0	0	0	1	0	3	0	0	0	3	0	0	1	0	0	1	1	5	2	7	24	0.785
	误报	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	正确	6	2	3	2	1	1	1	2	1	1	2	3	4	11	3	4	0	0	0	0	2	49	
	漏报	0	0	0	1	1	0	1	0	0	1	0	0	1	0	0	2	3	0	0	5	5	20	
	Manual	0	0	0	1	2	0	4	0	0	1	3	0	1	1	0	2	4	1	5	7	12	44	

4.2 实验设计与实现

对于避免修复副作用影响的问题, 我们需要根据测试路径生成测试用例, 覆盖受补丁修改影响的程序元素. 我们使用测试用例生成工具 Randoop4.3.1[20] 生成测试输入, 将运行时间设置为 3 min.

我们在人类编写的正确补丁上运行新的测试输入, 并将测试输出作为测试预言. 我们总是删除该过程中运行失败的测试用例 (失败的测试包括运行过程报

错、运行超时以及输出错误等), 之后用符号执行技术[21]对测试用例进行筛选, 删除重复项以得到最小的测试用例集.

我们在修复工具生成的补丁上执行这些测试用例. 如果补丁使任何测试用例失败, 则将其评估为过拟合. 对于第 2 个数据集, 我们收集原始正确程序来筛选测试用例集并运行测试套件以生成测试预言, 从而达到人类编写的正确的补丁的测试效果. 并且, 我们根据 QuixBugs 数据集中的纯随机测试方法对 BuggyJavaJML 中的补丁程序进行同样的纯随机测试, 以进行对比实验.

4.3 实验结果与分析

本节将分析我们的方法评估过拟合补丁的结果. 我们将其与手动评估以及其他 3 种技术的结果进行比较. 最后, 我们讨论了漏报和误报的案例.

表 2、表 3 第 1 行列出了由至少一个修复工具修复的错误程序的名称 (表格中为程序名称的字母缩写). 第 2 行显示所有工具生成的补丁程序总数. 中间部分分别显示了每种自动评估技术识别的过拟合补丁数量、错误地将正确补丁识别为过拟合补丁的误报数量、识别的正确补丁数量以及未将过拟合补丁识别出来的漏报数量. 我们在最后一行显示了手动评估的结果, 并基于该结果检测自动评估技术的准确性. 最后, 使用式 (7) 计算评估技术的准确性 (见表 2、表 3 最后一列).

$$\text{准确率}=\frac{\text{过拟合}+\text{正确}}{\text{过拟合}+\text{误报}+\text{正确}+\text{漏报}} \tag{7}$$

观察表 2、表 3, 我们可以发现 Our method 评估过拟合补丁的准确率高于 $\text{RGT}_{\text{InputSampling}}$. 这表明 Our method 的性能优于纯随机测试生成. 此外, Our method 比 $\text{GT}_{\text{Invariants}}$ 和 OpenJML 具有更高的准确性, 这是因为 Our method 比两者产生更少的误报 (QuixBugs: 0:12; BuggyJavaJML: 0:27).

虽然 Our Method 和 $\text{RGT}_{\text{InputSampling}}$ 在两个基准集上都不存在误报, 但是在两个数据集中 Our method 比 $\text{RGT}_{\text{InputSampling}}$ 分别多分类 25 个和 15 个过拟合补丁. 相反, $\text{GT}_{\text{Invariants}}$ 和 OpenJML 分别有 12 个和 27 个误报.

上述数字之间的差异可以通过他们收集的信息种类来解释, 并且受到异常值的影响. 比如, $\text{GT}_{\text{Invariants}}$ 可以捕获测试用例特定的值而不是取值范围. 例如可以捕获 x=0 而不是 $x\leqslant 0$, 这一现象背后的原因是因为测试用例只有 x=0 的情况, 此时 $\text{GT}_{\text{Invariants}}$ 捕获的不变量本身就是过拟合的, 因此 $\text{GT}_{\text{Invariants}}$ 虽然可以识别更多的过拟合补丁, 但是准确率不高.

在 BuggyJavaJML 数据集上, OpenJML 对过拟合补丁的分类比其他两种更多. 这主要是因为它将 27 个正确补丁分类为过拟合补丁. 如示例 5, ArjaE 在修复缺陷时会对程序进行不必要的修改. 我们观察到, 该补丁除了修复原始缺陷之外, 还增加了一句不必要的赋值语句“*fact*=*fact***c*”, 该语句因为插入在 JML 指定为纯的方法中造成了副作用, 因此从模块化验证的角度来看程序不正确, OpenJML 因此产生误报.

示例 5. ArjaE 为 Factorial-bug2 生成的补丁

```
for (c = 1; c < n; c++)
  fact = fact*c;
  + fact=fact*c;
return fact;
```

通过分析表 2、表 3, 我们发现 $\text{RGT}_{\text{InputSampling}}$ 的漏报最多 (QuixBugs: 26; BuggyJavaJML: 20). 这表明随机生成测试输入进行补丁评估的有效性较弱.

我们讨论了一个只被 Our method 正确识别为过拟合补丁的案例. 如示例 6, 该修复位置位于 for 循环中, 生成的修复程序使用“$(n \% n) == 0$”代替“$i < max$”. 手动评估显示, 此修补程序过拟合: 如果 n=0, 则 Cardumen 生成的修复程序和标准补丁的行为不同, 程序产生的输出也不同. 修复后的程序在 n 取特定值 0 时会抛出 java.lang.ArithmeticException 异常. 由于 $\text{RGT}_{\text{InputSampling}}$ 无法生成此特定输入, 因此无法正确识别该补丁是否为过拟合补丁.

示例 6. Cardumen 为 GET_FACTORS 生成的补丁

```
@@ –16, 7 +16, 7 @@
return new ArrayList<Integer>();
int max = (int)(Math.sqrt(n) + 1.0);
– for (int i=2; i < max; i++)
+ for (int i = 2; (n % n) == 0; i++)
if (n % i == 0)
ArrayList<Integer> prepend = new ArrayList<Integer>(0);
```

5 结论

本文研究了因测试套件不充分而引发的过拟合问题, 提出了一种基于数据流分析的过拟合补丁识别方法. 本文方法在两个数据集上进行了评估, 可以过滤 97.2% 和 94.6% 的过拟合补丁. 实验结果表明, 所提出的过拟合补丁识别方法可以有效避免修复副作用的影响, 进而识别更多的过拟合补丁. 在未来的工作中, 我

们将考虑回归测试最小化和选择排序技术, 对测试用例进行优先级排序, 从而加快补丁正确性评估.

参考文献

1 姜佳君, 陈俊洁, 熊英飞. 软件缺陷自动修复技术综述. 软件学报, 2021, 32(9): 2665–2690. [doi: 10.13328/j.cnki.jos.006274]

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.

3 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]

4 Kou R, Higo Y, Kusumoto S. A capable crossover technique on automatic program repair. Proceedings of the 7th International Workshop on Empirical Software Engineering in Practice (IWESEP). Osaka: IEEE, 2016. 45–50. [doi: 10.1109/IWESEP.2016.15]

5 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 34th International Conference on Software Engineering (ICSE). Zurich: IEEE, 2012. 3–13.

6 Le Goues C, Nguyen T, 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]

7 Long F, Rinard M. An analysis of the search spaces for generate and validate patch generation systems. Proceedings of the 38th International Conference on Software Engineering. Austin: IEEE, 2016. 702–713.

8 王赞, 郜健, 陈翔, 等. 自动程序修复方法研究述评. 计算机学报, 2018, 41(3): 588–610. [doi: 10.11897/SP.J.1016.2018.00588]

9 Yu XD, Wei FG, Ou XM, *et al*. GPU-based static data-flow analysis for fast and scalable Android APP vetting. Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). New Orleans: IEEE, 2020. 274–284. [doi: 10.1109/IPDPS47924.2020.00037]

10 Putra IS, Rukmono SA, Perdana RS. Abstract Syntax Tree (AST) and Control Flow Graph (CFG) construction of notasi algoritmik. Proceedings of the 2021 International Conference on Data and Software Engineering (ICoDSE). Bandung: IEEE, 2021. 1–6. [doi: 10.1109/ICoDSE53690.2021.9648437]

11 Schrettner L, Jász J, Gergely T, *et al*. Impact analysis in the presence of dependence clusters using static execute after in WebKit. Journal of Software: Evolution and Process, 2014, 26(6): 569–588. [doi: 10.1002/smr.1614]

12 Ye H, Martinez M, Durieux T, *et al*. A comprehensive study of automatic program repair on the QuixBugs benchmark. Journal of Systems and Software, 2021, 171: 110825. [doi: 10.1016/j.jss.2020.110825]

13 Nilizadeh A, Leavens GT, Le XBD, *et al*. Exploring true test overfitting in dynamic automated program repair using formal methods. Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation (ICST). Porto de Galinhas: IEEE, 2021. 229–240.

14 Qi YH, Mao XG, Lei Y, *et al*. The strength of random search on automated program repair. Proceedings of the 36th International Conference on Software Engineering. Hyderabad: ACM, 2014. 254–265.

15 Durieux T, Monperrus M. Dynamoth: Dynamic code synthesis for automatic program repair. Proceedings of the 11th International Workshop in Automation of Software Test. Austin: IEEE, 2016. 85–91.

16 Debroy V, Wong WE. Using mutation to automatically suggest fixes for faulty programs. Proceedings of the 3rd International Conference on Software Testing, Verification and Validation. Paris: IEEE, 2010. 65–74.

17 White M, Tufano M, Martinez M, *et al*. Sorting and transforming program repair ingredients via deep learning code similarities. Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Hangzhou: IEEE, 2019. 479–490.

18 Yang B, Yang JQ. Exploring the differences between plausible and correct patches at fine-grained level. Proceedings of the 2nd IEEE International Workshop on Intelligent Bug Fixing (IBF). London: IEEE, 2020. 1–8.

19 Arcuri A, Iqbal MZ, Briand L. Random testing: Theoretical results and practical implications. IEEE Transactions on Software Engineering, 2012, 38(2): 258–277. [doi: 10.1109/TSE.2011.121]

20 Pacheco C, Ernst MD. Randoop: Feedback-directed random testing for Java. Proceedings of the Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion. Montreal: ACM, 2007. 815–816.

21 Cadar C, Godefroid P, Khurshid S, *et al*. Symbolic execution for software testing in practice: Preliminary assessment. Proceedings of the 33rd International Conference on Software Engineering. Honolulu: IEEE, 2011. 1066–1071.

(校对责编: 孙君艳)