笔记模板2

1. 文章解决的问题

背景：从过去的补丁中收集到的变化来作为补丁候选生成的变化（数据）越来越多，当然也扩展了搜索空间。本文是收集了细粒度的抽象语法树的变化（更改）来扩展搜索空间。文章从过去的补丁中不断地获取更改，然后扩大自己的更改池。但是在更改池中找到特定的更改也越来越困难，所以他提出了使用程序以及补丁对应的上下文来寻找更改。而且上下文也对定位选择有帮助，针对defects4j，它可以过滤60%左右的bug候选语句。

它提出从bug程序中找到生成补丁所需的具体变量名称的可能性更大，而从bug程序中找到补丁所需要的成分可能性更小。

1. 使用变更上下文来查找收集到的补丁空间中的变更
2. 检查上下文在bug语句选择的用处，可以过滤一些可疑的bug语句
3. 通过名称选择来评估变更具体化。还利用类型兼容性来帮助具体化的更改

2. 解决的思路

第二章：CCA原理

1. 更改收集（change collect）

1. 变化提取以及归一化

   CCA通过收集AST更改来缓解具体代码风格更改带来的不同。其中用到的技术是Kim在2016年提出的技术

```
//Change 1
public void updateLabel(Node n){
  n = findNode(id);
- n.setLabel(label);
+ if(n != null) {
+   n.setLabel(label);
+ }
//Change 2
- sb.append(nodeType);
+ sb.append(nodeLabel);
```

   经过ast提取后得到的更改为

– **insert** t: insert a subtree t.
– **delete** t: delete a subtree t.
– **move** t: move a subtree t.

```
//Change 1
replace
  n.setLabel(label);
with
  if(n != null) {
    n.setLabel(label);
  }
//Change 2
update
  nodeType
to
  nodeLabel
```

(a) Extracted AST Subtree Changes.

```
//Change 1
replace
  var0.method0(var1);
with
  if(var0 != null) {
    var0.method0(var1);
  }
//Change 2
update
  var0
to
  var0
```

(b) Normalized Changes.

虽然会将具体的变量进行更改，使其抽象，但是对于java标准库中的一些变量、方法以及特定值是不会更改的。而且还会保留它的变量类型信息，为之后的变量具体化匹配作准备。

对于change2中的var0到var0是因为对于ast子树来说nodeType是旧的子树，nodeLable是新的子树，所以不会对其他子树的变量产生影响。也就是说只有在同一子树中，不同的变量会有不同的抽象变量。

2. 改变上下文识别（Change Context Identification）

   从变更子树节点的附近来表示ast上下文

   这个的好处，举例：

```
public void method() {  //Parent
......
//Change 1
     n = tree.getNode(id); //Left
-    label = n.getLabel();
+    if(n != null)
+      label = n.getLabel();
     sb.append(label); //Right
//Change 2
     n = new Node(type); //Left
-    label = n.getLabel();
+    if(n != null)
+      label = n.getLabel();
     sb.append(label); //Right
......
}
```

比较两个空指针的检查，可以发现1是有意义的，2是无意义。如果用来上下文，那么就可以避免2的情况。

更改的上下文可以由节点父P、左L、右R来表示。P代表更改的代码所属的代码块（有整行的代码也可能是一个方法或变量，具体 需要看所选的操作）。L代表更改代码前面的代码片段，R指之后的代码片段

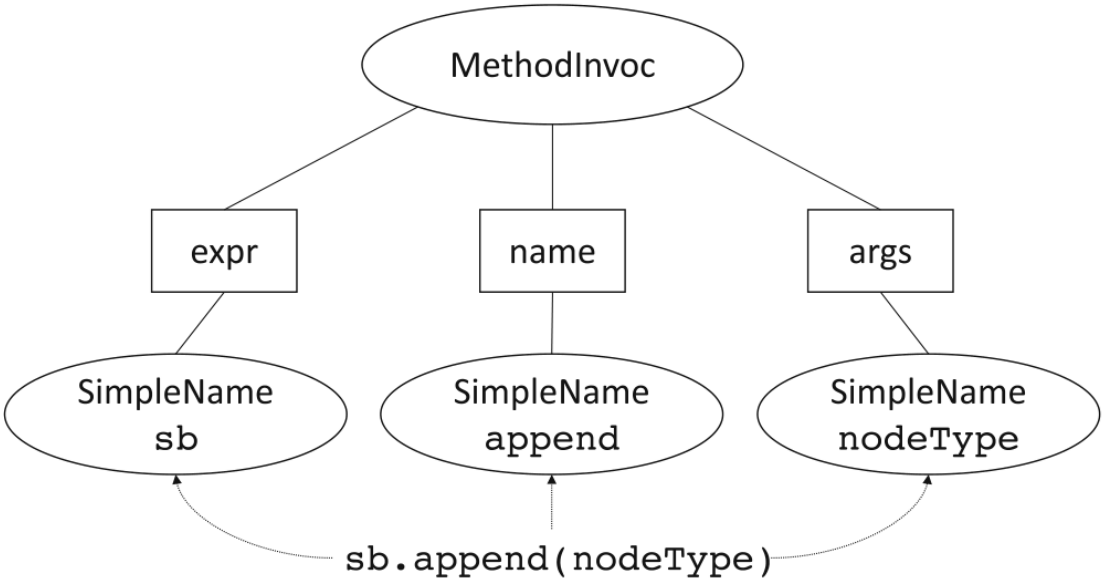
CCA设计了一种更抽象的上下文节点类型。（通过哈希函数）节点类型由两部分组成：ast节点类型（AST node type）和语法位置（syntactic location）

AST节点类型:

$$t(n,c) = \begin{cases} n.type[c.loc], & \text{if } n \text{ is the parent of } c \\ n.type[n.loc], & \text{otherwise} \end{cases}$$

c指的是更改的子树的根。c.loc指的是c的语法位置。也就是说当修饰父节点P的时候，父节点用的nodetype使用c这个更改的语法位置的。

如果修饰R或L的话，就是用自生的语法位置。



比如上述例子，MethodInvoc指的是方法调用（append），但是更改的是nodeType,所以MethodInvoc实际的loc是nodeType的loc，即args，那么P的节点类型的表达方式为MethodInvoc[args]。

节点hash:

这个用来区分上下文的不同，使用的技术是Dyck word hashing (Chilowicz et al.2009)

$$h(n) = \{n.type\{h(c_1), h(c_2), \cdots, h(c_k)\}\},$$
$$\text{where } c_1, c_2, ..., c_k \text{ are child nodes of } n$$

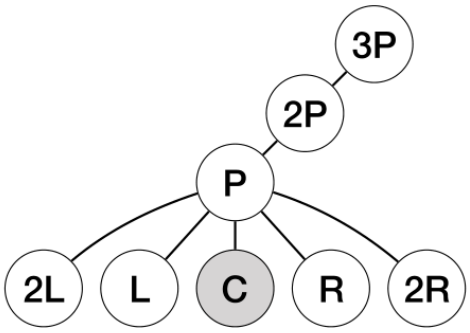
```
public void method() {  //Parent
......
//Change 1
     n = tree.getNode(id); //Left
-    label = n.getLabel();
+    if(n != null)
+      label = n.getLabel();
     sb.append(label); //Right
//Change 2
     n = new Node(type); //Left
-    label = n.getLabel();
+    if(n != null)
+      label = n.getLabel();
     sb.append(label); //Right
......
}
```

如果只用node type，那么对于前面的图就无法区分change1和2的L节点了，因为都是赋值语句。用hash可以对两个L节点进行分辨。

对于L1：Assignment{SN,MI{SN,SN,SN}}, 对于L2： Assignment{SN,CIC{ST,SN}}。

SN：simpleName，MI：方法调用， CIC：类实例创建， ST： simpleType

再来一个例子，说明精度的变化



(a) Change Contexts

```
public void method() {   // 3P
  while(condition1) { // 2P
    if(condition2) { // P
      statement1; // 2L
      statement2; // L
+       inserted; // C
      statement3; // R
      statement4; // 2R
    }
  }
}
```

(b) Code Example

例如（P,2P,3P）可以扩大或缩小ast中的更改精度

接下去是上下文的九种类型：

Group	Name	Parent (C_P)	Left (C_L)	Right (C_R)
parent	PT	t(P)		
	2PT	t(P),t(2P)		
	3PT	t(P),t(2P),t(3P)		
type	PLRT	t(P)	t(L)	t(R)
	P2LRT	t(P)	t(L),t(2L)	t(R),t(2R)
	2P2LRT	t(P),t(2P)	t(L),t(2L)	t(R),t(2R)
hash	PTLRH	t(P)	h(L)	h(R)
	PT2LRH	t(P)	h(L),h(2L)	h(R),h(2R)
	2PT2LRH	t(P),t(2P)	h(L),h(2L)	h(R),h(2R)

总结

当提取更改且识别上下文之后，更改会被存储在更改池中，并且按照上下文分类。

2 生成补丁

总的步骤就是缺陷定位，然后按照可疑度排列。将bug语句的上下文都列出来，与更改池中的进行查找。查找成功后用name selection来将具体的变量一一对应。再生成补丁

1. Fix Location Selection

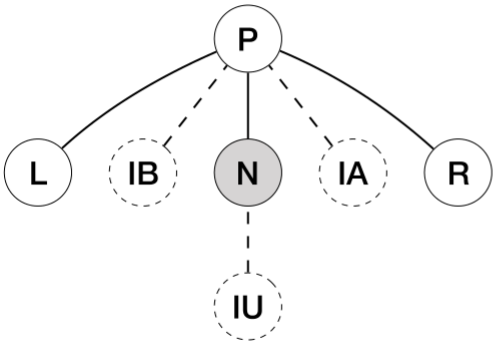
   这一步是为了识别可疑AST节点的位置上下文。并且要将这个上下文与更改池中的上下文兼容。

   有四种位置上下文：

   - Default： 节点N 删除、替换、修改
   - Insert Before： 节点N之前插入一个子树（具体就是前面插入一条语句）
   - Insert After： 节点N之后插入一个子树

- Insert Under：在节点N的下面插入一个子树（方法体里面插入一条语句）

每个可疑语句的节点都会产生四个上述的位置上下文。根据SBFL技术来对bug节点进行排序

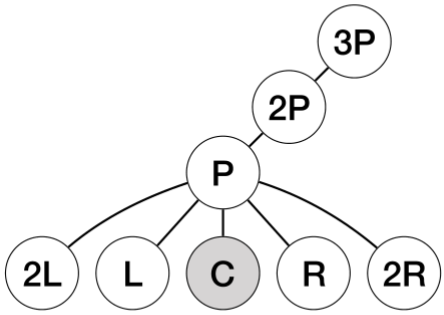


(a) Location Contexts

```
public void method() {  // P
......
  statementX; // L
+ inserted before // IB
  covered;  // N
+ inserted after  // IA
  statementY; // R
......
  while(condition1) { // N
+     inserted under  // IU
  }
}
```

(b) Code Example

图片1



(a) Change Contexts

```
public void method() {  // 3P
  while(condition1) { // 2P
    if(condition2) { // P
      statement1; // 2L
      statement2; // L
+       inserted; // C
      statement3; // R
      statement4; // 2R
    }
  }
}
```

(b) Code Example

图片二

对于图2来说，b是修改后的程序，inserted是插入的新语句。源程序是没有inserted的。那么当statement2作为图片一中的covered的节点时，statement2的位置上下文应该选IA（insert after）。

其中insert的操作是特殊的。原程序中没有inserted，那么c应该是statement2或statement3.然后用c的上下文来匹配更改池里的上下文

怎么在四个位置上下文选择正确的一个？：

利用上下文频率：CCA根据人类手写的补丁对四个位置上下文进行统计，计数较高的位置上下文会被CCA优先选择

2. change selection（更改选择）

在更改池中查找符合fix location的上下文的更改。首先查找到相同的上下文。然后再对相同上下文下的更改进行查找。

查找有三种算法：

- Random：随机选择一种
- Roulette-Wheel Selection：根据更改的频率作为概率来随机查找
- Most Frequent First : 根据更改出现的频率降序排序，然后每次选择首个更改，产生一个补丁后。再次选择下一个更改

3. Change Concretization and Name Selection

在更改池中选择了更改后就需要将它具体化，就是将bug程序中的具体变量赋值到更改中。

有四种名称选择方法：

- Random：随机选择，可能导致编译失败。
- Type-Compatible：通过更改池中的保存的类型信息来匹配变量、方法、赋值方法
- Variable-oriented Fault Localization(VFL):利用Kim提出的方法来匹配变量。通过失败的测试用例执行的语句中的变量被认为是更加与bug语句相关的，更相关的变量就更可取。
- Hybrid Method (TC-VFL):将前两者结合起来

3. 核心知识点或名词定义

CCA: Context-based Change Application

搜索空间：基于搜索的算法都有的搜索空间

该方法的抽象操作有五种：

1. insert t: insert a subtree t
2. delete t: 删除一个子树
3. move：移动一个子树
4. update n1, n2: n1 to n2
5. replace ： 将t1 转为t2

4.程序功能说明

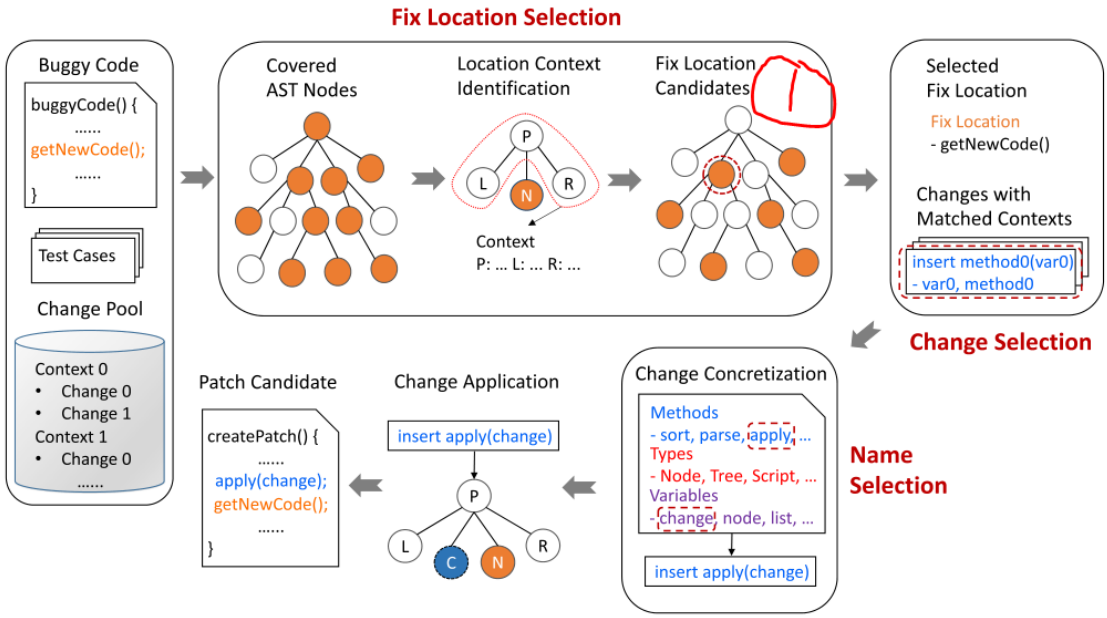


Fig. 6 The overview of patch generation of a repair model using context-based change application technique

在fix location selection中对每一个可疑的节点生成上下文后，在更改池中查找上下文，如果没有找到则将这个节点排除。

5. 存在的问题

6. 改进的思路

7. 想法来源

1. Martinez et al. proposed a repair model based on repair actions collected from human-written patches (Martinez and Monperrus2015).：提出基于人类手写的补丁的修复的修复模型
2. Nguyen et al. investigated change repetitiveness and tested change recommendation for bug-fixes (Nguyen et al.2013).

3. There are other studies about recurrence of code fragments and changes which imply the potential of ideas using previous patches for new patch generation (Barr et al.2014; Gabel and Su2010; Ray et al.2014; Martinez et al.2014; Zhong and Su2015; Zhong and Meng 2018)
4. Some studies discussed about search space and its navigation issue (Long and Rinard 2016a; Q i e t a l .2015; Smith et al.2015; L e e t a l .2018).