标识符重命名不一致性缺陷的检测

王 伟，苏小红，马培军，王甜甜

（哈尔滨工业大学 计算机科学与技术学院，150001 哈尔滨，wwmuou@sina.com）

摘 要：为解决已有的克隆代码及相关缺陷检测工具无法分析大型程序代码，又不能识别经过修改了的克隆代码的问题，在改进基于频繁子序列挖掘的克隆代码检测模型基础上，提出了基于序列挖掘的 *C* 克隆代码及标识符重命名不一致性缺陷检测模型. 该模型改进了已有的忘记修改某标识符缺陷检测子模型，并增加了错误修改某标识符缺陷检测子模型，通过计算标识符未改变率来检测标识符重命名不一致性缺陷. 实验表明，该模型能在克隆代码检测的基础上检测克隆代码引起的相关缺陷，降低了漏检率和误检率，适用于2种标识符重命名不一致性缺陷的检测.

关键词：序列挖掘；克隆代码；标识符重命名不一致性；软件缺陷检测

中图分类号：TP311 **文献标志码**：A **文章编号**：0367－6234(2011)01－0089－06

Detection of identifier renaming inconsistency

WANG Wei, SU Xiao-hong, MA Pei-jun, WANG Tian-tian

(School of Computer Science and Technology, Harbin Institute of Technology, 150001 Harbin, China, wwmuou@sina.com)

Abstract: For solving the problem that the current clone and related bug detection tools can not analyze large-scale code and detect modified code clones, a model of clone and identifier inconsistency bugs detection based on the improved model of clone detection via sequential pattern mining is proposed. This model improves the existing submodel of identifier forgotten renaming bug detection, and introduces a new submodel of identifier mistaken renaming bug detection. Identifier renaming inconsistency bugs are detected by computing identifier unchanged ratio (UnchangedRatio). Experiment results indicate that this model can detect clone related bugs on the basis of code clone detection, reduce false-negative rate and false-positive rate, and is effective in detecting two kinds of identifier inconsistency bugs.

Key words: sequential pattern mining; code clone; identifier inconsistency bug; software bug detectiion

拷贝－粘贴手段的使用使得大型软件中克隆代码(Duplicated Code，或 Code Clone)的比例很高. 拷贝粘贴后常因修改而引入缺陷. 然而大多软件缺陷检测工具仅能识别缓冲区溢出、内存泄露等缺陷，不易检测出由拷贝－粘贴代码引起的语义缺陷. 常用的克隆代码检测方法有：基于字符串[1－5]、基于 Token 流[6－9](如 CCFinder)、基于抽象语法树[10－14]和基于程序语义的方法[15－16]. 其中，基于抽象语法树和基于程序语义的方法，时间复杂度过高，不适用于大型软件；基于字符串和基于 Token 流的方法适用于分析大型软件，但是不能识别经过增、删、改操作修改了的克隆代码，相关工具如 CCFinder 等不能识别由克隆代码引起的软件缺陷. 文献[17]将序列模式挖掘算法与克隆代码检测相结合，降低了时间复杂性，还能检测“拷贝－粘贴－修改”行为导致的“忘记修改某标识符”的软件缺陷. 但误检率很高，而且不能检测因“错误修改某标识符”导致的标识符重命名不一致性缺陷.

针对上述问题，本文提出 Token 流分析和序列挖掘相结合的方法，将克隆代码和克隆代码引起的缺陷进行联合检测，降低克隆代码误检和漏

收稿日期：2009－12－19.

基金项目：国家自然科学基金资助项目(61073052)；高等学校博士学科点专项科研基金资助项目(20092302110040).

作者简介：王 伟(1986—)，男，硕士研究生；
苏小红(1966—)，女，教授，博士生导师；
马培军(1963—)，男，教授，博士生导师.

检对标识符重命名不一致缺陷检测的影响. 在此基础上,改进了“忘记修改某标识符”缺陷检测子模型,同时增加了“错误修改某标识符”缺陷检测子模型,有效降低了漏检率和误检率.

1　改进的基于频繁子序列挖掘的克隆代码检测模型

原基于频繁子序列挖掘克隆代码检测模型[18]在克隆代码检测时会发生较高的误检和漏检,经过分析发现:

1) 不在同一函数范围内的代码碎片被合并成为一个重复代码片段导致的误检;

2) 相同类型的不同标识符转换成相同的Token导致的误检;

3) 克隆代码稀疏不连续导致的误检;

4) 克隆代码对中对应位置标识符不同的对数较多导致的误检;

5) 输入的 C 源程序中,相同的语句分不同行数表示导致的漏检;

6) 碎片合并过程中合并 - 删除策略不完善导致的漏检;

7) 代码拷贝粘贴过程中因错误修改某标识符而导致的漏检.

为了解决上述问题,本文改进了该模型,改进后的流程如图1所示.

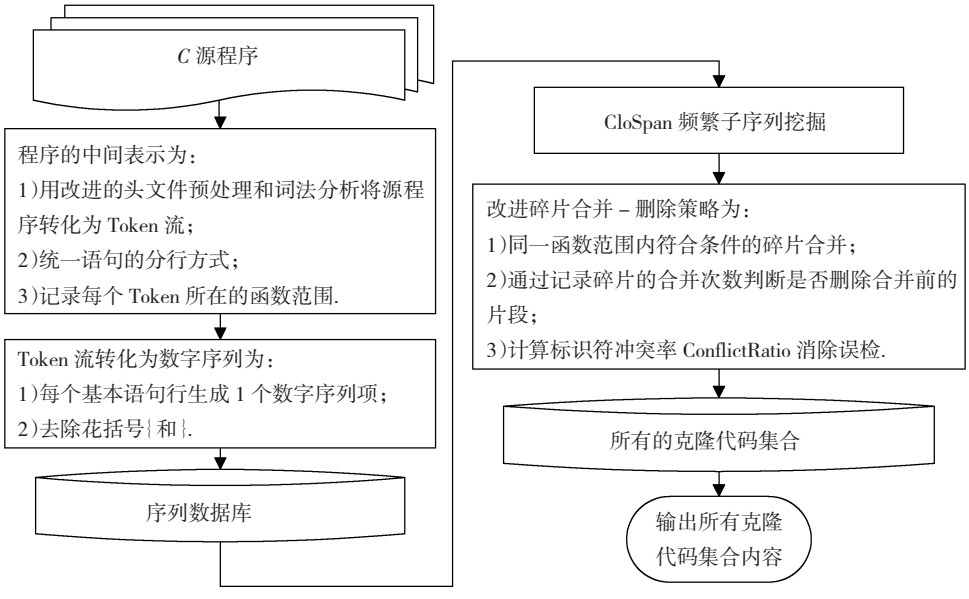

图1　改进的基于频繁子序列挖掘的克隆代码检测模型的流程

1) 在划分基本模块,生成数字序列阶段,依据序列的函数范围信息进行同一函数范围内的碎片合并,解决不同函数范围内的碎片合并带来的误检.

2) 在创建序列数据库时,由于算法将所有类型相同的标识符(变量,函数名等)映射成相同的Token值,不考虑它们实际的名字,这种宽松条件允许克隆代码片段中的标识符重命名. 然而,该过程可能会产生误检,为此利用剪枝方法,引入标识符映射冲突率,即通过计算代码对的标识符映射冲突率来判断是否发生了误检,从而降低误检. 文献[17]形式化地描述了克隆代码对的标识符映射等相关概念,并给出了标识符映射冲突率(ConflictRatio)的计算公式.

3)序列挖掘中允许序列中插入 gap,因此可以识别插入、删除或修改了语句的克隆代码片段,但也带来了误检. 如图2所示,gap 值设为3,由原始序列挖掘出的克隆代码过于稀疏,不符合拷贝粘贴的习惯,因此判定为误检. 本文加入了比例控制,即限定插入、删除或修改了的语句总行数不超过克隆代码行数的1/2.

4) 为了能够检测出个别标识符被修改后的克隆代码片段,在词法分析阶段将标识符映射为同一个 Token,但这也导致了图3所示的误检,“结构相同,但标识符大多不同”,不符合拷贝 - 粘贴的习惯,而且这类误检在误检中占有很大的比重. 本文定义了结构一致误检率来解决此类问题.

5) 在词法分析阶段,统一语句的分行方式,将每个最基本的语句陈述作为一行,解决相同的语句分不同行数表示导致的漏检.

6) 改进碎片“合并 - 删除”策略. 为每个克隆代码集合的每个片段设置一个合并次数变量,

```
 - 0038 - L0001              int func1 ( )
 - 0039 - L0002              {
1- 0040 - L0003 - CL0001 -     int a , b , c ;
2- 0041 - L0004 - CL0001 -     a = 3 ;
3- 0042 - L0005 - CL0001 -     b = a ++ ;
4- 0043 - L0006 - CL0001 -     c = - a ;
 - 0044 - L0007                if ( b == c )
 - 0045 - L0008                {
 - 0045 - L0009                   return 1 ;
 - 0045 - L0010                }
 - 0046 - L0011                return 0 ;
 - 0047 - L0012              }
```

```
 - 0049 - L0013              int func2 ( int m )
 - 0050 - L0014              {
 - 0051 - L0015                 int n ;
1- 0052 - L0016 - CL0013 -      int a , b , c ;
 - 0053 - L0017                 n = m ;
 - 0054 - L0018                 n ++ ;
 - 0055 - L0019                 n ++ ;
2- 0056 - L0020 - CL0013 -      a = 3 ;
3- 0057 - L0021 - CL0013 -      b = a ++ ;
 - 0058 - L0022                 ++ n ;
 - 0059 - L0023                 ++ n ;
4- 0060 - L0024 - CL0013 -      c = - a ;
 - 0061 - L0025                 return n ;
 - 0062 - L0026              }
```

图 2　克隆代码稀疏不连续导致的误检

```
        spin_lock_init ( & chip->lock);
-CP 1-  chip->card = card;
-CP 2-  chip->read = read;
-CP 3-  chip->write = write;
-CP 4-  chip->private_data = private_data;
        init_timer(&chip->timer);
```

```
        strcpy( timer->name,"s");
-CP 1-  chip->private_data = chip;
-CP 2-  chip->private_free = snd_cs4231_timer_free;
-CP 3-  chip->hw = snd_cs4231_timer_table;
-CP 4-  chip->timer = timer;
        return 0;
```

图 3　克隆代码对中对应位置标识符不同的对数较多时导致的误检

用来记录每个片段参与过的合并的次数. 当合并次数达到某一设定的上限时,再将该片段删除.

7) 增加了对"错误修改某标识符"导致的标识符重命名不一致性缺陷的检测.

对于给定的疑似拷贝 - 粘贴代码对$< F_1, F_2 >$,设标识符 $A \in F_1$,A 在 F_1 中出现的所有位置的集合表示为 $\text{Place}_1(A) = \{PA_1, PA_2, \cdots, PA_i\}$.

定义 1(克隆代码对的标识符映射)　对于给定的克隆代码对 $< F_1, F_2 >$ 和 F_1 中的某标识符 $A(A \in F_1)$,将 A 从 F_1 到 F_2 的标识符映射表示为一个集合 $\text{Mapping}(A)$,这个集合是 F_2 中与 $\text{Place}_1(A)$ 对应位置出现的标识符的集合 $\text{Mapping}(A) = \{A'_1, A'_2, \cdots, A'_k\}$.

定义 2(一致的标识符映射)　对于给定的克隆代码对$< F_1, F_2 >$和F_1中的某标识符$A(A \in F_1)$,称 A 的标识符映射是一致的,如果 A 总是映射到 F_2 中的同一个标识符,即 $| \text{Mapping}(A) | = 1$.

定义 3(不一致的标识符映射)　对于给定的克隆代码对 $< F_1, F_2 >$ 和 F_1 中的某标识符 $A(A \in F_1)$,称 A 的标识符映射是不一致的,如果 A 映射到 F_2 中 2 个或 2 个以上不同的标识符,即 $| \text{Mapping}(A) | > 1$.

为了识别标识符不一致的情况,通过计算代码对的标识符冲突率来判断是否发生了误检. 标识符冲突率的计算公式为

$$\text{ConflictRatio}(< F_1, F_2 >) = \sum_{A \in \text{All Identifiers in } F_1 \& F_2} \text{Weight}(A) \times \text{ConflictRatio}(A). \quad (1)$$

其中,

$$\text{Weight}(A) = \frac{\text{Occurrence}(A \text{ in } F_1 \text{ or } F_2)}{\text{Occurrence}(\text{All Identifiers of } F_1 \& F_2)} \leqslant 0.5. \quad (2)$$

每个标识符的映射冲突率的计算公式为

$$0 \leqslant \text{ConflictRatio}(A \in F_1) = 1 - \frac{\max\{\text{Occurrence}(A') \mid A' \in \text{Mapping}(A)\}}{\text{Occurrence}(A \in F_1)} < 1. \quad (3)$$

如果 ConflictRatio($< F_1, F_2 >$) 大于某个预设的阈值,说明克隆代码对 $< F_1, F_2 >$ 中存在许多标识符不一致的情况,那么很可能不是真正的克隆代码,则判断为误检.

定义 4(克隆代码对的标识符映射对)　对于给定的克隆代码对 $< F_1, F_2 >$,称片段 F_1 和片段 F_2 对应位置的标识符对为克隆代码对 $< F_1, F_2 >$ 的标识符映射对,所有标识符映射对的集合组成标识符映射对集(doubleMapping(F_1, F_2)),$\text{doubleMapping}(F_1, F_2) = \{(a_1, b_1), (a_2, b_2), \cdots, (a_n, b_n)\}$,其中 $a_i \in F_1, b_i \in F_2$.

通过计算结构一致误检率(structureRatio),来消除克隆代码对中对应位置标识符不同的对数较多时导致的误检. 结构一致误检率的计算公式为

$$\text{structureRatio}(F_1, F_2) = \frac{| \text{doubleMapping}(F_1, F_2) | a_i \neq b_i |}{| \text{doubleMapping}(F_1, F_2) |}. \quad (4)$$

当 structureRatio(F_1, F_2) 大于设定的阈值时,说明克隆代码对 $< F_1, F_2 >$ 中对应位置标识符不同的对数占总对数的比例较大,不符合正常

的拷贝 - 粘贴习惯,因此判定为误检.

2 标识符重命名不一致性缺陷检测

2.1 标识符重命名不一致性缺陷检测模型

克隆代码检测的目的是为了解决与克隆代码相关的问题. 本文关注的是在克隆代码检测的基础上,检测其相关的软件缺陷. 在改进的基于频繁子序列挖掘的克隆代码检测模型基础上,提出了基于序列挖掘的 C 克隆代码及标识符重命名不一致性缺陷检测模型,如图4所示. 该模型由改进的基于频繁子序列挖掘的克隆代码检测子模型和标识符重命名不一致性缺陷检测子模型组成.

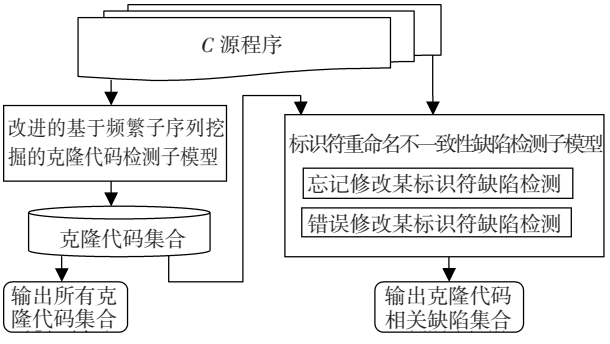

图4 基于序列挖掘的 C 克隆代码及相关软件缺陷检测模型

2.2 "忘记修改某标识符"的标识符重命名不一致性缺陷检测

开发人员复用某段代码时,通常是在拷贝 - 粘贴后进行修改,从而实现自己想要的功能. 如果源代码片段中某标识符在拷贝 - 粘贴代码中的所有出现位置上被修改成了2个或2个以上不同的名称,就出现了标识符重命名不一致的现象,称其为克隆代码的标识符重命名不一致性.

克隆代码的标识符重命名不一致性通常伴随着软件缺陷,克隆代码中导致软件缺陷的标识符重命名不一致性为标识符重命名不一致性缺陷. 如果源代码片段的某个标识符 A,在拷贝 - 粘贴代码的所有出现位置,除了一、两处跟原名称相同,剩下的位置都被重命名为另一个名称 B,就意味着很可能是开发人员想把该标识符 A 的所有出现位置都改为 B,但是忘记修改了某处,这样就产生了标识符重命名不一致性缺陷.

文献[17]率先在克隆代码标识符映射不一致性的基础上提出了"忘记修改某个标识符"的标识符重命名不一致性缺陷的检测方法. 对于一个在源代码片段中出现多次的标识符而言,如果在拷贝 - 粘贴代码片段中的对应位置上总是映射到相同的标识符,则是一致的,反之,则是不一致的. 获得一个克隆代码对的所有标识符的映射关系后,可以找到映射不一致的标识符,其中就可能存在标识符重命名不一致性缺陷.

标识符映射不一致不一定说明存在缺陷,如果不一致数量很高,则2个代码片段很可能是误检的克隆代码片段. 因此,关键的问题是判断一对存在标识符映射不一致性的克隆代码是否存在缺陷. 为了解决这个问题,文献[17]通过计算克隆代码对的标识符未改变率(UnchangedRatio)来检测在标识符映射中的缺陷. UnchangedRatio 的计算公式为

$$\text{UnchangedRatio} = \frac{\text{NumUnchanged}}{\text{NumTotal}}. \tag{5}$$

式中:NumUnchanged 为一个给定标识符在拷贝 - 粘贴代码中没有被重命名的出现次数;NumTotal 为该标识符在源代码片段中出现的总次数;UnchangedRatio 的值为 0 ~ 1. 当一个标识符的 UnchangedRatio = 1 时,表示该标识符在拷贝 - 粘贴代码中的所有出现均未被重命名,UnchangedRatio = 0 表示该标识符在拷贝 - 粘贴代码中的所有出现均被重命名. 因此,当 UnchangedRatio ≠ 0 时,UnchangedRatio 越小,意味着标识符保留标识符名的位置越少,越可能是"忘记修改某个标识符"的软件缺陷. 如果某个标识符的 UnchangedRatio 低于规定的某阈值,则认为该标识符名称未改变的出现位置是一个缺陷.

结合标识符映射相关概念,给出了标识符未改变率 UnchangedRatio 的计算公式为

$$\text{UnchangedRatio}(A \in F_1) = \frac{\text{Occurrence}(A \in \text{Mapping}(A))}{\text{Occurrence}(A \in F_1)}. \tag{6}$$

由于标识符重命名不一致性缺陷检测是通过克隆代码片段两两比对来实现的,因此在文献[17]中,当同一个克隆代码组内片段较多,且其中的一个片段存在误检时,会导致该误检以组合的方式出现,进而导致较高的误检率. 针对该问题,本文首先找出带有缺陷的片段,对于每一个带有缺陷的片段,只保留第1个包含该缺陷的克隆代码片段对,并对带有缺陷的片段进行标记显示,方便了开发人员对该缺陷的确认和修复.

2.3 "错误修改某标识符"的标识符重命名不一致性缺陷检测

在研究中,除了忘记修改某标识符引起的缺陷外,发现还存在另外一种标识符重命名不一致性缺陷. 这种缺陷是由在拷贝 - 粘贴代码后,错误的将某个标识符重命名为与其他标识符相同的名

字而导致的. 例如，如果源代码片段的某个标识符 A，在拷贝 - 粘贴代码的所有出现位置，除了一、两处被重命名为 B，剩下的更多位置的标识符名依然为 A. 这意味着很可能是开发人员在重命名其他标识符时，不小心将此处的 A 错误地重命名了. 本文将这种缺陷称之为“错误修改某个标识符“的标识符重命名不一致性缺陷.

```
算法：BuildCnfletRatioArray
输入：克隆代码数组（CP_Seg_Array）
输出：检测出的标识符重命名不一致性缺陷
Begin
    for each 克隆代码片段对 <F1,F2> in CP_Seg_Array
        if 克隆代码的行数 < Min_len then
            continue;
        endif
        建立 <F1,F2> 的标识符映射集合;
        for each 标识符 in F1 or F2
            计算每个标识符的标识符未改变率（UnchangedRatio）;
            if 0 < UnchangedRatio ≤ UnchangedRatioThreshold then
                加入“忘记修改某标识符”缺陷列表;
            else if 0 < 1 - UnchangedRatio ≤ UnchangedRatio-
Threshold then
                加入“错误修改某标识符”缺陷列表;
            endif
        endfor
    endfor
    输出标识符重命名不一致性缺陷信息到文本;
End
```

图 5　克隆代码的标识符重命名不一致性缺陷检测算法

标识符重命名不一致性缺陷检测算法如图 5 所示. 首先定义一个阈值 UnchangedRatioTheshold. 当 UnchagedRatio≠0 时，UnchagedRatio 越小，说明标识符保留原名的个数越少，$<F_1,F_2>$ 越可能存在“忘记修改某标识符”的标识符重命名不一致性缺陷，当 0 < UnchagedRatiod ≤ UnchangeRatiodThreshold 时，判定为存在“忘记修改某标识符”的标识符重命名不一致性缺陷；当 UnchagedRatio≠1 时，UnchagedRatio 越大，说明标识符保留原名的个数越多，$<F_1,F_2>$ 越可能存在“错误修改某标识符”的标识符重命名不一致性缺陷，当 0 < 1 - UnchagedRatiod ≤ UnchangedRatioThreshold 时，判定为“错误修改某标识符”的标识符重命名不一致性缺陷.

3　结果与分析

首先，对 Linux2. 6. 6 源代码 arch，net，kernel，crypto 模块进行标识符重命名不一致性缺陷检测（设标识符未改变率阈值 UnchangedThreshold 为 0. 4），检测出的实际缺陷数如表 1 所示。

其次，采用人工注入缺陷的方式，对 Linux2. 6. 6 和 httpd2. 2. 2（下载自 http://www. apache. org/）的部分源代码测试的实验结果如表 2，3 所示。

表 1　对实际开源代码进行标识符重命名不一致性缺陷检测的结果

子模块	C 文件数	源代码行数	克隆代码组数	克隆代码行数	检测出的实际缺陷数	
					忘记修改某标识符	错误修改某标识符
arch	2 353	719 415	5 534	48 914	3	27
net	536	333 741	2 543	19 017	4	9
kernel	47	30 629	103	703	0	0
crypto	25	9157	52	674	0	0

表 2　忘记修改某标识符缺陷（bug1）实验结果

实验源代码及行数	未注入缺陷时对源代码检测出的缺陷个数	人工注入的缺陷个数	对已注入缺陷的源代码检测出的人工注入的缺陷个数（#Suspects）	确认的人工注入的缺陷个数（#Bugs）	对已注入缺陷的源代码检测出的源代码中原有的缺陷个数	人工注入的缺陷的漏检率 TP/%	人工注入缺陷的误检率 FP/%
linux - 2. 6. 6\sound\driver（12 380）	1	20	21	20	1	0	4. 8
linux - 2. 6. 6\arch\arm（49 480）	1	45	50	45	1	0	10
httpd - 2. 2. 2\srclib（126 966）	9 *	96	126	96	6 *	0	23. 8

注：* 为人工注入缺陷后，检测出的原有代码中的缺陷发生漏检.

表3 错误修改某标识符缺陷(bug2)实验结果

实验源代码及行数	未注入缺陷时对源代码检测出的缺陷个数	人工注入的缺陷个数	对已注入缺陷的源代码检测出的人工注入的缺陷个数(#Suspects)	确认的人工注入的缺陷个数(#Bugs)	对已注入缺陷的源代码检测出的源代码中原有的缺陷个数	人工注入的缺陷的漏检率TP/%	人工注入缺陷的误检率FP/%
linux-2.6.6\sound\driver(12 380)	1	20	20	20	1	0	0
linux-2.6.6\arch\arm(49 480)	6	45	49	45	6	0	8.2
httpd-2.2.2\srclib(126 966)	26*	96*	124	95*	24*	1.0	23.4

注:*为人工注入缺陷后,检测出的原有代码中的缺陷发生漏检.

其中,误检率计算公式为

$$FP=\frac{\#Suspects-\#Bugs}{\#Suspects}\times 100\%.$$

人工注入缺陷后,检测出的原有代码中的缺陷发生漏检,是由于源代码某个克隆代码片段中某个标识符存在不一致性缺陷,当人工再次注入一个缺陷后,该标识符被命名为2个以上不同的标识符.人工未注入缺陷前,片段2中的标识符vec存在一个不一致性缺陷,人工注入缺陷后,把片段2中的一个vec修改为另一个标识符(设置为vec2),导致片段1中的irq映射为3个标识符:vec、irq和vec2,考虑到映射为2个以上的标识符时忘记修改某标识符的可能性很小,所以程序中被作为误检情况"滤掉"了,因此造成漏检.

从实验结果可以看出,本文的模型对人工注入的缺陷具有较低的漏检率和误检率,适用于检测克隆代码相关的标识符不一致性缺陷.由于编程的灵活性和复杂性,检测到的缺陷均需开发人员确认.

4 结 论

1)改进了已有的克隆代码检测模型,降低了漏检率和误检率.

2)改进了忘记修改某标识符缺陷检测模型,降低漏检率和误检率的同时,方便了开发人员对可疑缺陷的审查.

3)增加了错误修改某标识符缺陷检测模型,来检测"错误修改某标识符"的标识符重命名不一致性缺陷.

参考文献:

[1] DUCASSE S, RIEGER M, DEMEYER S. A language independent approach for detecting duplicated code [C]//Proceedings of the IEEE International Conference on Software Maintenance. Washington. DC: ACM, 1999: 109-118.

[2] van RYSSELBERGHE F. Vingerafdrukken van code om duplicatie op te sporen [D]. Antwerpen: Master Thesis at Universiteit Antwerpen, 2002.

[3] BAKER B S. A program for identifying duplicated code [C]//Proceedings of Computing Science and Statistics 24th Symposium on the Interface Interface. New Jersey: Interface Foundation of North America, 1992: 132-141.

[4] BAKER B S. Parameterized duplication in strings: algorithms and an application to software maintenance [C]//Proceedings of the 25th Annual ACM Symposium on Theory of Computing. New York, NY: ACM, 1993: 75-96.

[5] BAKER B S. Parameterized duplication in strings: Algorithms and an application to software maintenance [J]. SIAM Journal on Computing, 1997, 26(5): 1343-1362.

[6] CHURCH K W, HELFMAN J I. Dotplot: a program for exploring selfsimilarity in millions of lines of text and code [J]. Journal of Computational and Graphical Statistics, 1993, 2(2): 153-174.

[7] KAMIYA T, KUSUMOTO S, INOUE K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code [J]. IEEE Transacitions on Software Engineering, 2002, 28(7): 654-670.

[8] BASIT H A, SMYTH W F, TURPIN A, *et al*. Efficient token based clone detection with flexible tokenization [C]//Proceeding ESEC-FSE Companion '07 The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers. New York, NY: ACM, 2007: 513-516.

(下转第98页)