文章编号：2095-2163(2020)03-0175-06 中图分类号：TP391 文献标志码：A

语句包裹模式的识别和分类

于永胜，杨春花
（齐鲁工业大学（山东省科学院）计算机科学与技术学院，济南 250353）

摘 要：软件演化过程中会产生大量变更代码，对变更代码的识别有利于变更理解。其中普遍存在的把一个语句或语句序列移动到一个或多个不同的语法实体中的变更行为，对于这种语句包裹模式的识别和分类，提出了基于代码变更块和抽象语法树的语句包裹模式识别分类算法。首先从变更前后版本 2 个文件中筛选出代码变更块，根据语句包裹模式的特征找到候选代码变更块，再建立抽象语法树，通过语法分析找到代码变更块中存在的语句包裹模式并对其进行分类。该算法在 4 个开源项目中进行了实验验证，实验结果表明该算法对语句包裹模式的识别具有较高的准确率。

关键词：语句包裹模式；软件演化；抽象语法树；代码变更块

Identification and classification of statement encapsulation patterns

YU Yongsheng, YANG Chunhua
(School of Computer Science and Technology, Qilu University of Technology (Shandong Academy of Sciences), Jinan 250353, China)

【Abstract】 During the software evolution process, a large number of change codes will be generated, and the identification of the change code is conducive to change understanding. Among them is the universal behavior of changing a statement or a sequence of statement into one or more different grammatical entities. For the recognition and classification of such statement encapsulation patterns, the paper proposes a statement encapsulation pattern recognition classification algorithm based on code change blocks and abstract syntax trees. First, the code change block is filtered from the two files before and after the change, and the candidate code change block is found according to the characteristics of the statement encapsulation pattern, then an abstract syntax tree is established. Finally, syntactic analysis is used to find and classify the statement encapsulation patterns existing in the code change block. The algorithm has been experimentally verified in four open source projects, and the experimental results show that the algorithm has a high accuracy rate for statement encapsulation pattern recognition.

【Key words】 statement encapsulation pattern; software evolution; abstract syntax tree; code change block

0 引 言

软件版本控制系统记录着软件开发过程中的每个细节以及不同时期的不同版本，软件版本控制系统每天都会产生大量的变更代码，理解变更代码可以使软件日常维护和功能添加变得更加容易。理解软件变更已经成为软件开发人员日常开发和维护的基本要求。

在大量软件变更代码中存在对软件的重构、错误修复以及功能添加。重构是在不改变软件外在行为的前提下，改进软件的内部结构，使软件更易于维护和功能的添加。Fowler[1] 在其著作中详细介绍了重构模型，其核心是一个全面的重构目录。变更代码中也存在对代码的错误修复。关于错误报告方面的研究有：Zhang 等人[2] 对 Bugzilla 上的错误报告进行了实证研究，并指出了进行错误报告分析时可能存在的问题；童燕翔[3] 去除了 bug report 中的非修正性报告并提出了自动化缺陷定位方法；关于错误修复的识别方面的研究有：Eshkevari 等人[4] 提出了沿 4 个正交维度的标识符重命名分类方法，并实现了标识符重命名的识别和分类；Kawrykow 等人[5] 开发了一种工具支持的技术，用于检测软件系统的修订历史记录中不必要的代码修改；Kim 等人[6] 通过分析错误修复的历史记录开发了错误修复存储器，并提出了一种使用错误修复存储器的错误发现算法；关于错误预测方面的研究有：原子等人[7] 提出了语句级的缺陷预测方法，揭示了变更易于引入缺陷的因素；刘望舒等人[8] 针对挖掘软件历史仓库过程中程序模块类型标记和软件度量时产生的噪声，

基金项目：山东省自然科学基金面上项目（ZR2017MF056）。

作者简介：于永胜（1992-），男，硕士研究生，主要研究方向：代码变更分析、软件演化；杨春花（1974-），女，博士，教授，主要研究方向：代码变更分析、软件演化、软件开发。
通讯作者：于永胜 Email：2607563830@qq.com

收稿日期：2020-01-06

提出一种可容忍噪声的特征选择框架 FECS。

软件变更代码中掺杂着大量的对软件的重构、错误修复和功能添加的代码,对软件变更代码中变更模式的提取有利于对软件变更的理解。在对软件变更进行研究后,接着又研究了把一个语句或语句序列移动到一个或多个不同的语法实体中的语句包裹模式。Pan 等人[9]在其著作中从 7 个开源项目的配置管理存储库中,使用错误修复程序更改所涉及的语法组件和源代码上下文定义了 27 个可自动提取的错误修复模式。但是其著作对错误修复模式的研究并没有完全涵盖语句包裹模式所包含的变更类型,因此还需要对语句包裹模式做进一步的研究。

1 语句包裹模式的识别和分类

1.1 代码变更块

代码变更块(hunk)是由代码变更前后版本之间的变更代码构成的,本文通过基于文本的方法获取代码变更块,图 1 中的代码变更块取自数据仓库 jEdit 中 KeymapImpl.java 文件的一次提交。在图 1 所示的代码变更块中,左边页码从第 106~107 行前面带减号的代码是代码变更过程中从原文件中删除的代码,右边页码从第 106~111 行前面带加号的代码是代码变更后新版本文件中增加的代码。

```
106     -       modified = true;
107     -       props.setProperty(name, shortcut);
    106 +       String oldShortcut = props.getProperty(name);
    107 +       if (!shortcut.equals(oldShortcut))
    108 +       {
    109 +           modified = true;
    110 +           props.setProperty(name, shortcut);
    111 +       }
```

图 1 hunk 示意图

Fig. 1 hunk diagram

1.2 抽象语法树

抽象语法树抽象地表示了源代码的语法结构,抽象语法树通过将源代码中的每一种结构用节点表示来表现编程语言的语法结构,而真实语法中的每一个细节并没有包含在抽象语法树中,比如嵌套的括号在抽象语法树中就没有以节点的形式表现出来,对于语句包裹模式的研究需要对代码变更文件的抽象语法树进行分析。

1.3 语句包裹模式的识别和分类

1.3.1 语句包裹模式

语句包裹模式所表示的代码变更行为是把一个语句或语句序列移动到一个或多个不同的语法实体中,这个语法实体是一个结构化的语句,语法实体包括 If、Try、For、Switch、Synchronized、While、Labeled 和 Do。语句包裹模式所表示的代码变更行为是将一个语句或语句序列移动到一个或多个结构化语句中,这种结构化语句是这个语句或语句序列在被执行时需要满足的条件,是对代码原文件的错误修复。

在图 2 所示的语句包裹模式 hunk 实例取自数据仓库 jEdit 中文件 Gutter.java 的一次提交中,图 2 中左边页码第 128 行是从原文件中删除的代码,右侧页码从 128~129 行是代码变更过程中增加的代码,第 128 行删除的代码和第 129 行增加的代码是非常相似的,图 2 中将一行代码移动到了一个 If 语法实体中,发生了语句包裹。对于一个语句或语句序列移动到 Try、For、Switch、Synchronized、While、Labeled 和 Do 这七种语法实体中的变更行为也都属于语句包裹模式。

```
128     -           updateLineNumberWidth();
    128 +           if (numLines != 0)
    129 +               updateLineNumberWidth();
```

图 2 语句包裹模式 hunk 实例

Fig. 2 Statement encapsulation pattern hunk instance

在图 3 所示的语句包裹模式的 hunk 实例取自数据仓库 jEdit 中文件 EditPane.java 的一次提交,第 1236 行删除的代码和第 1214 行增加的代码是非常相似的,就是将第 1236 行代码从一个 If 语法实体中移动了出来,发生了逆向的语句包裹,属于语句包裹模式。

```
1235    -       if(_buffer == buffer)
1236    -           textArea.propertiesChanged();
    1214 +      textArea.propertiesChanged();
```

图 3 逆向的语句包裹模式 hunk 实例

Fig. 3 Reverse statement encapsulation pattern hunk instance

1.3.2 语句包裹模式的识别和分类

假设存在 h_a 和 h_d 分别为 hunk 中增加的代码和删除的代码,l_a 和 l_d 分别为 hunk 中增加代码和删除代码的相似部分,则 p_r 和 p_l 分别为 l_a 和 l_d 的所有父节点,c_r 和 c_l 分别为 l_a 和 l_d 在 hunk 内的所有父节点,m_r 和 m_l 分别是 l_a 和 l_d 父节点中的第一个方法节点,当这次变更属于语句包裹模式时,则需要满足以下条件:

(1)语句包裹模式的候选 hunk 中需要存在语句或语句序列移动到一个或多个语法实体中,这些移动的语句或语句序列是相似的,所以 hunk 中需要存在相似部分。

(2)hunk 中相似部分的父节点中第一个方法节点 m_r 和 m_l 的方法名是相同的。对于 l_a 和 l_d 在 hunk 内的父节点 c_r 和 c_l,需要满足一个为空集且另一个不为空集的条件,且 l_a 和 l_d 的所有父节点 p_r 和 p_l 不能完全相同。

判断此次变更属于语句包裹模式后,当 c_r 不为

空且 c_l 为空时，可以判断变更属于正向的语句包裹模式，并根据语法实体的种类对语句包裹模式进行分类，这里的语法实体就是 c_r 的第一个节点。当 c_l 不为空且 c_r 为空时，可以判断为逆向的语句包裹模式，将 c_l 的第一个节点作为语法实体，并根据这里的语法实体对语句包裹模式进行分类。

1.3.3　根据代码相似度识别 hunk 中的代码移动

在图 2 语句包裹模式的候选 hunk 中，从原版本文件中删除的代码是左侧页码的第 128 行，代码变更后增加的代码是右侧页码的第 128～129 行，原版本文件中第 128 行代码在代码变更后移动到了新版本文件的第 129 行，这些移动的语句或语句序列具有很高的相似度，因此要识别这样的代码移动需要对变更代码进行相似度识别。

1.3.4　通过抽象语法树识别语句包裹模式

从语句包裹模式候选 hunk 中找到移动的语句或语句序列后，需要确定 hunk 相似部分 l_a 和 l_d 的父节点 p_r 和 p_l 中的第一个方法节点 m_r 和 m_l 的方法名需要相同，还需要确定 hunk 和相似部分之间存在一个或多个语法实体，也就是 l_a 和 l_d 在 hunk 内的父节点 c_r 和 c_l 需要一个为空集一个不为空集，l_a 和 l_d 的所有父节点 p_r 和 p_l 不能完全相同。

首先获取 l_a 和 l_d 的行范围 r_a 和 r_d，获取 hunk 增加代码和删除代码 h_a 和 h_d 的行范围 k_a 和 k_d，遍历代码变更前后版本的两个文件的语法树，找到行范围分别包含 r_a 和 r_d 的父节点 p_r 和 p_l，找到行范围位于 r_a 和 k_a 之间的父节点 c_r，找到行范围位于 r_d 和 k_d 之间的父节点 c_l，分别从 p_r 和 p_l 中找到 l_a 和 l_d 的第一个方法节点 m_r 和 m_l，判断 m_r 和 m_l 是否是同名函数节点，判断 c_r 和 c_l 是否一个为空集、另一个不为空集，判断 p_r 和 p_l 是否不完全相同，如果都满足，则该代码变更是语句包裹模式。

1.3.5　语句包裹模式识别和分类算法

本算法输入的是变更前后 2 个版本的文件 $file_l$ 和 $file_r$，输出是语句序列移动到的语法实体 E_n 和该语法实体的分支 E_b，$E_n = < e_n >$，e_n 是 $file_l$ 和 $file_r$ 文件中的一个语句包裹模式变更中的语句或语句序列移动到的语法实体，$E_b = < e_b >$，e_b 是语句或语句序列移动到语法实体 e_n 中后所位于的 e_n 的分支。

算法 1　语句包裹模式识别算法

输入：代码变更前后版本两个文件 $file_l$ 和 $file_r$

输出：$file_l$ 和 $file_r$ 文件中每一个语句包裹模式变更中语法实体 e_n 的集合 E_n 和语句移动到语法实体后所在的分支 e_b 的集合 E_b

```
1(H_a, H_d) ← getHunk(file_l, file_r)
2  (L_a, L_d) ← getSimilarPart(H_a, H_d)
3  K_a ← getHunkRange(H_a)
4  K_d ← getHunkRange(H_d)
5  t_l ← generateAST(file_l)
6  t_r ← generateAST(file_r)
7  R_a ← getSimilarPartRange(L_a)
8  R_d ← getSimilarPartRange(L_d)
9  (P_r, C_r, B_r) ← ergodicAST(K_a, R_a, t_r)
10  (P_l, C_l, B_l) ← ergodicAST(K_d, R_d, t_l)
11 for each c_l ∈ C_l, c_r ∈ C_r, p_l ∈ P_l, p_r ∈ P_r, b_l ∈ B_l, b_r ∈ B_r
12   m_l ← getMethodNode(p_l)
13   m_r ← getMethodNode(p_r)
14   if(isSimilarMethod(m_l, m_r))
15     if ((c_l = ϕ)&&(c_r ≠ ϕ) ‖ ((c_l ≠ ϕ)&&(c_r = ϕ)
16       if(p_l ≠ p_r)
17        if((c_l ≠ ϕ)&&(c_r = ϕ))
18          e_n ← identifyEncapsulateNode(c_l)
19          e_b ← identifyBranch(b_l)
20          E_n ← e_n
21          E_b ← e_b
22        else if ((c_l = ϕ)&&(c_r ≠ ϕ))
23          e_n ← identifyEncapsulateNode(c_r)
24          e_b ← identifyBranch(b_r)
25          E_n ← e_n
26          E_b ← e_b
27        end if
28       end if
29     end if
30   end if
31 end for
```

算法第 1 行生成代码变更前后版本文件的 hunk 集 H_a 和 H_d，从 hunk 集中识别出相似部分的集合 L_a 和 L_d，第 3 ～ 4 行得到 hunk 集 H_a 和 H_d 的行范围集合 K_a 和 K_d，第 5 ～ 6 行得到 $file_l$ 和 $file_r$ 的抽象语法树的根节点 t_l 和 t_r，第 7 ～ 8 行得到 hunk 的相似部分的集合 L_a 和 L_d 的行范围集合 R_a 和 R_d，第 9 ～ 10 行通过遍历 $file_r$ 和 $file_l$ 的语法树得到相似部分的所有父节点集合 P_r 和 P_l、在 hunk 内的父节点的集合 C_r 和 C_l 以及相似部分在父节点 P_r 和 P_l 中每一个节点的分支的集合 B_r 和 B_l，第 11 ～ 13 行分别找到 p_r 和 p_l 中相似部分的第一个满足是方法节点的父节

点,第 14 行判断 2 个方法父节点是否满足方法名不相同的条件,第 15 行判断相似部分在 hunk 内的父节点 c_l 和 c_r 是否满足一个为空集、另一个不为空集的条件,第 16 ~ 27 行判断相似部分在文件中的所有父节点 p_l 和 p_r 是否满足不完全相同的条件。算法的第 17 ~ 21 行中,当 c_l 和 c_r 满足 c_l 不为空集而 c_r 为空集时,将 c_l 中第一个节点作为语句包裹模式的语法实体存入 e_n 中,e_n 存入 E_n,并将该语法实体的相应分支 b_l 存入 e_b 中,e_b 存入 E_b。第 22 ~ 31 行中,当 c_l 为空集而 c_r 不为空集时,将 c_r 的第一个父节点作为语句包裹模式的语法实体存入 e_n,e_n 存入 E_n,并将该语法实体的相应分支 b_r 存入 e_b 中,e_b 存入 E_b。

上述算法中第 9~10 行使用了函数 *ergodicAST*,下面是对函数 *ergodicAST* 的描述:函数 *ergodicAST* 的输入为文件 hunk 集 H 的行范围的集合 K,hunk 集中相似部分 L 的行范围的集合 R,以及文件的语法树的根节点 t。输出为文件中相似部分集合的所有父节点的集合 P,相似部分在 hunk 集内的父节点的集合 C,以及相似部分在父节点集合 P 中每一个节点的分支的集合 B。

算法 2 *ergodicAST* **算法**

输入:文件 hunk 集 H 的行范围 K,相似部分集 L 的行范围 R 以及文件的语法树的根节点 t

输出:文件 hunk 集相似部分的父节点集合 P,相似部分在 hunk 集内的父节点集合 C,相似部分在父节点集合 P 中每个节点的分支的集合 B

```
1  for each k ∈ K, r ∈ R
2    N ← t
3    while (N ≠ ϕ)
4      n ← takeOutFirstNode(N)
5      p ← n
6      r_z ← getNodeRange(n)
7      if(r ⊆ r_z) && (r_z ⊆ k)
8        c ← n
9      end if
10     B_z ← getAllBranch(n)
11     for each b_z ∈ B_z
12       N_t ← getAllNode(b_z)
13       r_b ← getRange(N_t)
14       if( r ⊆ r_b)
15         b ← b_z
16       end if
17       for each n_c ∈ N_t
18         r_c ← getNodeRange(n_c)
19         if( r ⊆ r_c)
20           N ← n_c
21         end if
22       end for
23     end for
24   end while
25   P ← p
26   C ← c
27   B ← b
28 end for
```

算法第 1~5 行将文件语法树中相似部分的第一个父节点存入 p 中,第 6~9 行判断该节点是否也在 hunk 内,如果在 hunk 内就将其存入 c 中,第 10~16 行获取 n 节点的分支集合 B_z,并将包含相似部分的分支存入 b 中,第 17 ~ 24 行遍历分支内的节点,将包含相似部分的节点存入 N,若 N 不为空集,则继续遍历文件的语法树。第 25 ~ 28 行将 p、c 和 b 分别存入 P、C 和 B 中。

2 算法的实现和验证

本文实验使用了 Java 编程语言、基于文本的差异化分析工具 Gnu Diff[10] 以及 eclipse jdt parser 工具。首先使用了基于文本的差异化分析工具 Gnu Differ(http://www.gnu.org/software/diffutils)获取代码变更前后版本文件的代码变更块 hunk,再使用 eclipse jdt parser 工具分析代码变更前后版本文件的抽象语法树。并从 4 个开源项目中获取了数据集用于实验研究。

2.1 数据获取

本实验使用了从 4 个开源项目中获取的数据集,下面是对本实验使用的数据集的介绍。

(1)开源项目 jEdit(http://sourceforge.net/project/jedit),该开源项目是一个跨平台的文本编辑器。

(2)开源项目 eclipseJDTCore(http://www.eclipse.org/downloads/eclipse-packages/),该开源项目是一个面向 Java 的集成开发环境。

(3)开源项目 maven(http://maven.apache.org/),该开源项目是通过信息描述来管理项目构建、报告和文档的管理工具。

(4)开源项目 google-guice(https://github.com/apress/google-guice),该开源项目是一个轻量级的依赖注入容器。

表 1 是描述了从 4 个开源项目中获取的数据集的项目名称、获取时间段、提交总数、文件总数以及提交和文件的匹配数 commit_file 的信息。

表 1　4 个数据集中数据的获取时间段、提交总数、文件总数以及提交和文件的匹配版本总数

Tab. 1　Time period of data acquisition, total number of submissions, total number of files, and total number of matching versions of submissions and files in the four datasets

项目名称	时间段	提交总数	文件总数	提交和文件的匹配数 commit_file
jEdit	1998/9~2012/8	6 486	2 943	36 195
eclipse	2001/6~2013/10	19 321	5 815	89 030
maven	2003/9~2014/1	9723	4 327	40 686
google-guice	2006/8~2013/12	1 198	1 371	40 309

2.2　实验结果及分析

本文中分别从 4 个开源项目的数据集中人工抽取了语句包裹模式变更集，并使用本文算法在人工抽取的变更集上进行了实验验证，实验结果见表 2。

表 2　本文算法在人工抽取的语句包裹模式变更集上进行验证后的实验结果

Tab. 2　The experimental results of the algorithm in this paper are verified on the artificially extracted sentence encapsulation mode change set

项目名称	人工抽取的语句包裹变更实例的数量	算法识别的语句包裹模式实例的数量	识别率/%
jEdit	70	60	86
eclipse	90	78	87
maven	70	63	90
google-guice	50	45	90

从实验结果可以看出，本文算法在人工抽取的语句包裹模式的变更集中的识别率在 86%~90%之间波动。

将本文算法在 4 个开源项目的数据仓库中进行实验验证后，从实验结果所有版本中抽取了 10%进行人工检测，对从实验结果中抽取的 10%进行人工检测后的验证结果见表 3。

表 3　从实验结果集中抽取 10%进行人工检测后的验证结果表

Tab. 3　A table of verification results after extracting 10% from the experimental result set for manual testing

项目名称	算法检测的语句包裹的总版本数	抽取算法检测结果的 10%	人工核对结果	准确率/%
jEdit	2 089	209	204	98
eclipse	5 073	507	482	95
maven	1 237	124	121	98
google-guice	190	19	18	95

本文算法对 4 个开源项目数据仓库进行了语句包裹模式识别，对从识别结果所有版本中抽取的 10%进行人工检测后，检测结果的准确率在 95%~98%之间波动。

本文算法在 4 个开源项目数据仓库上进行实验得到实验结果集后，从结果集中抽取每一种语句包裹模式的变更版本集的 10%进行人工检测，实验结果见表 4。

表 4　对从实验结果集中的每种语句包裹模式版本集中抽取 10%进行人工检测后的验证结果表

Tab. 4　The verification result table after manually detecting 10% of each sentence encapsulation pattern version set from the experimental result set for manual detection

项目名称	抽取算法识别结果所有版本的 10%								人工检测的为真的版本								准确率/%
	If	Try	For	Switch	Synchronized	While	Labeled	Do	If	Try	For	Switch	Synchronized	While	Labeled	Do	
jEdit	179	23	12	3	2	3	1	1	173	22	12	3	2	3	1	1	97
eclipse	417	59	26	21	12	8	4	3	400	57	24	19	12	7	4	3	96
maven	87	31	13	1	1	1	1	0	85	30	12	1	1	1	1	0	97
google	11	6	2	1	0	0	0	1	11	5	2	1	0	0	0	1	95

对从实验结果集中每种语句包裹模式变更版本集中抽取 10%并进行人工检测后，实验结果显示准确率在 95%~97%之间波动。

对人工抽取的变更集的识别率以及对从结果集中抽取实例进行人工验证的准确率未能达到 100%的原因，在于语句包裹粒度较大的语句包裹模式变更被 Gnu Diff 划分到了多个 hunk 中，这种情况降低了本文算法识别结果的准确度，将来可以为大粒度语句包裹模式变更提供 hunk 合并功能，以提高对于这种情况的识别准确率。

3　结束语

本文通过对代码变更的研究，介绍了代码变更中普遍存在的语句包裹模式，提出了基于代码变更

（下转第 182 页）