保持语义不变的C克隆代码预处理方法

边奕心[1,2]，赵 松[2]，杜 军[2]

(1. 中国科学院软件研究所 北京 海淀区 100190；2. 哈尔滨师范大学计算机科学与信息工程学院 哈尔滨 150025)

【摘要】克隆代码检测工具的输出结果由于存在克隆检测不一致性缺陷的误检和检测出的克隆代码不能直接用于重构的问题，需要对检测工具的输出结果进行预处理。为了解决该问题，提出一种新的克隆代码预处理方法。首先，将自适应K-最近邻聚类方法与程序依赖图相结合，用于降低克隆不一致性相关缺陷检测的误检。然后，使用基于代价-收益分析的评估方法，在消除缺陷后的克隆代码中识别可重构的克隆代码。实验结果表明，该预处理方法，不仅降低了克隆不一致性相关缺陷检测工具产生的误检，提高了可重构克隆代码的数量，而且将克隆代码检测与克隆代码重构两个过程连接为一个有机的整体，有利于提高软件的质量，降低软件维护的成本。

关 键 词 自适应K-最近邻聚类； 克隆代码； 克隆不一致性缺陷检测； 程序依赖图； 重构

中图分类号 TH133; TP183 **文献标志码** A doi:10.3969/j.issn.1001-0548.2017.06.023

Semantic-Preserving Pre-Processing Method for C Clone Code

BIAN Yi-xin[1,2], ZHAO Song[2], and DU Jun[2]

(1. Institute of Software Chinese Academy of Sciences Haidian Beijing 100190;

2. College of Computer Science and Information Engineering, Harbin Normal University Harbin 150025)

Abstract The output results of clone code detection tool cannot be directly refactored because of the two reasons: one is the false positives of clone inconsistency related bugs detection and the other is that all the detected clones cannot be suitable for refactoring. Therefore, the output results of clone code detection tool need to be pre-processed for reducing the error checking of cloning inconsistencies defect. A pre-processing approach combing adaptive K-nearest neighbor clustering with program dependence graph is proposed in this paper to solve these problems. First, adaptive K-nearest neighbor clustering and program dependence graph are used to reduce the false positives of clones inconsistency related bugs detection. And then the refactorable clone code is identified to reduce the cost of clone maintenance. The results of the study show that our approach not only effectively prunes the false positives of clone inconsistency related bugs but also eliminates the gap between clone code detection and clone refactoring. Therefore, our method contributes to improving the quality of the software and decreasing the cost of software maintenance.

Key words adaptive K-nearest neighbor clustering; clone code; clone inconsistency related bugs detection; program dependence graph; refactoring

克隆代码(clone code)是程序源代码中多个具有相似语法或语义的代码片段，是程序员的拷贝–粘贴行为造成的[1]，被列为软件中低质量、难以理解、难以维护的代码中最著名的一种“坏味道(bad smell)”。研究显示，克隆代码增加了源代码的长度和软件系统的复杂性，使其更加难以维护，并可能在降低系统运行效率的同时，引入大量缺陷[2]。通过对代码进行重构可消软件系统中的部分克隆代码，提高程序的质量。但是，克隆代码检测工具的输出结果，一般情况下不能直接用于重构，原因如下：首先，本文使用的检测工具CPBugdetector[3]在检测克隆代码的同时，还能对拷贝–粘贴的相关缺陷进行检测。但是，工具的输出结果存在一定的误检，对拷贝–粘贴相关缺陷检测结果的准确性，及对后续克隆代码的重构结果都会产生负面影响。其次，并非所有检测出来的克隆代码都适于重构[4-6]。为解决以上问题，本文提出了一种预处理方法，该方法结合程序依赖图和自适应K-最近邻聚类两种方法，减少克隆不一致性相关缺陷检测的误检，然后，使用基于代价–收益分析的评估方法，在消除缺陷后的克

收稿日期：2016 – 09 – 14；修回日期：2017 – 05 – 18
基金项目：黑龙江省自然科学基金(F2016030)
作者简介：边奕心(1979 –)，女，博士，主要从事软件重构、程序分析、缺陷检测方面的研究.

隆代码片段中识别可重构的克隆代码，目的是降低维护克隆代码的代价。

1 相关研究

1.1 克隆不一致性相关缺陷检测存在的问题

1.1.1 拷贝–粘贴相关缺陷的检测

克隆代码通常是由开发人员的拷贝-粘贴行为产生的，这种行为极易引入缺陷。通过分析操作系统中的缺陷，文献[7]认为拷贝–粘贴是导致软件出现缺陷的一个重要原因。研究结果表明：在Linux driver/i20的一个源文件中，发现的35个bug中有34个是由代码的拷贝–粘贴导致的。

开发人员在编写程序时，由于实践需要，通常会对复制后的代码进行修改，这是导致克隆不一致性缺陷的主要原因。如，如果程序的源代码片段被拷贝后，在被拷贝片段的相应位置的标识符经过修改后，出现了两个或两个以上不同的名称，这种不一致性通常预示着软件缺陷，被称为标识符重命名不一致性缺陷。文献[8]把这种标识符重命名不一致性缺陷称之为“忘记修改某标识符”。文献[9]又提出了另一种标识符不一致性缺陷：“错误修改某标识符”。两种缺陷的含义如下：

忘记修改某标识符缺陷[8]：源代码片段被拷贝–粘贴后，如果原片段的某个标识符*M*在被拷贝片段的所有位置上，除了一两处为*M*，其余的位置都被重命名为*N*，这很可能是开发者由于忘记修改了某处而引入的缺陷。

错误修改某标识符缺陷[8]：源代码片段被拷贝–粘贴后，如果原片段的某个标识符*M*在被拷贝的片段的所有出现位置上，除了一两处被重命名为*N*，剩下位置的为*M*，这很可能是开发者在重命名其他位置的标识符时，不小心将此处的标识符*M*错误地重命名为*N*而引入的缺陷。

图1所示程序是在Linux 2.6.6版本中，由检测工具CPBugdetector检测出来的一个错误修改某标识符缺陷，其中，片段1的标识符“ops”在片段2中的映射出现了不一致(映射“ops”4次，映射“claimed”1次)。这种缺陷很可能是开发人员“错误修改某标识符”而引入的软件缺陷。

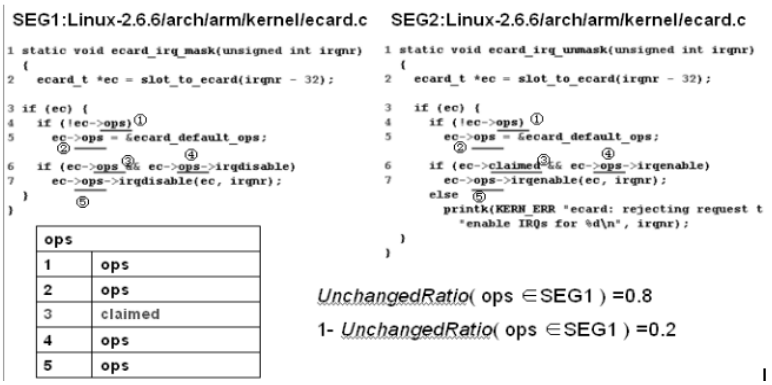

图1　从Linux2.6.6中检测出的一个错误修改某标识符缺陷

1.1.2 拷贝-粘贴相关缺陷检测的误检

代码中出现标识符映射的不一致并不意味着一定是缺陷，也有可能是拷贝–粘贴相关缺陷的检测工具造成的误检。因为到目前为止，还没有检测工具的输出结果是不存在误检的。原因可以分为3类：1) 错误匹配拷贝–粘贴片段；2) 语句顺序发生变换；3) 插入或删除结构相同的语句。前两种类型的误检出现在文献[8]中，但文献并没有提出误检消除的方法。本文主要研究后两种误检并给出具体的误检消除方法。本文前期的工作[10]对消除拷贝–粘贴相关缺陷检测的误检方法进行了初步研究，在此基础上，增加了可以消除的误检类型，并与度量方法相结合，在消除误检后的克隆代码中，识别可重构的克隆代码。

图2所示程序是第二种类型的误检——可交换的语句顺序。如图2b所示，片段1的标识符“lastdirent”映射到片段2中，发生了映射不一致，在片段2中该标识符被映射成“lastdirent”3次，一次被映射成标识符“error”。从缺陷检测结果看，这种映射的不一致是由于程序员错误修改某标识符而引起的缺陷。但是，通过分析可以发现，这个检测结果是由于片段1和片段2中存在可交换的语句顺序(语句15与16顺序颠倒)引起的，并不是真正的软件缺陷。

本文需要处理两种类型的误检：1) 插入或删除相同结构的语句；2) 可交换的语句顺序。这两种误检都是在原代码片段被拷贝粘贴后，在被拷贝片段

的出现位置的语句的顺序发生了变化引起的，为确定误检发生的位置(行号)，本文引入了自适应K-最近邻聚类方法。该方法通过计算实体(程序语句)之间的相似度，把功能相近的语句聚到一起，然后对源代码进行聚类，将关系最密切的实体(语句的行号)分到一个聚类集合中，以便对原代码片段和被拷贝片段的聚类集合中的行号及其相应语句的哈希值进行比较，确定误检发生的位置。

行号	片段 1	语句哈希值	行号	片段 2	语句哈希值
12	struct linux_dirent32* lastdire nt?	22356709	12	struct linux_dirent32* lastdire nt?	22356709
13	if(error < 0){	21501776	13	if(error < 0){	21501776
14	goto out_putf;{	105405920	14	goto out_putf;{	105405920
15	lastdirent ? =buf.previous	73109296		error ? = buf.error	73109296
16	error =buf.error;	73109296	15	lastdirent = buf.previous ;	73109296
17	if(lastdirent ?)	76781968	16	if(lastdirent ?)	76781968
18	put_user(file->file->f_pos,&lastdirent ?	86935968	17	put_user(file-> f_pos,&lastdirent ?	86935968
	->d_off);			->d_off);	
19	error= count-buf.count;}	66945165	18	error= count-buf.count;}	66945165

a. 误检

位置	片段 1	片段 2
?	lastdirent	lastdirent
?	lastdirent	error
?	lastdirent	lastdirent
?	lastdirent	lastdirent

b. 标识符映射关系

图2 从Linux 2.6.6 版本中检测出的一个语句位置互换引起的误检

1.1.3 识别克隆代码可重构机会的挑战

检测克隆代码是重构克隆代码的前提，但是，以下原因导致检测工具的输出结果并不都适于重构：首先，有些克隆代码是程序员的有意行为，因此，克隆代码不一定对软件系统产生威胁[4,11]，检测出来的克隆代码有多少会对软件系统产生不利影响，目前尚不清楚[5]。其次，由于大规模软件系统的程序复杂性，导致很难消除某些对系统存在真正威胁的克隆代码[6]。如，一些仅由声明语句组成的克隆代码是不适于提取的[12]。最后，有些克隆代码使用传统方法很难消除，如果采用特殊手段消除这些克隆代码，会导致代码难于理解，有时甚至有引入缺陷的可能性，这样的结果是克隆代码对系统的威胁仍然没有消除。本文前期研究[13]对克隆代码的识别方法进行了初步探索，在此基础上，提出了一种新的预处理克隆代码的方法。该方法将程序依赖图和自适应K-最近邻聚类相结合，消除或减少克隆不一致性相关缺陷检测结果中存在的误检，然后，使用基于代价–收益分析的评估方法，在消除缺陷的克隆代码中识别适于重构的克隆代码片段，在提高克隆代码识别安全性的同时降低克隆代码维护的代价。

1.2 预处理算法的相关理论基础

1.2.1 自适应K-最近邻聚类方法

自适应K-最近邻(adaptive K-nearest neighbor)聚类方法由文献[14]提出，用于处理多种问题。该方法通过计算实体(程序语句)之间的相似度，把功能相近的语句放到一个聚类集合中，以便重构。该方法与程序切片方法的思想类似，都是要找到相关度最高的语句，然后聚类。但是，程序切片方法需要先建立程序依赖图，然后分析语句间的各种依赖关系，算法的复杂度非常高，不能分析较大规模的程序。而聚类方法是通过计算实体(程序语句)之间的相似度把最相关的语句聚集到一起，计算简单，算法的复杂度低，适用于分析大规模的程序代码。

1.2.2 程序依赖图

程序依赖图(program dependence graph, PDG)是一种程序的中间图形表示形式，是一个带有标记的有向多重图。文献[15]提出程序依赖图的概念，并给出建图的详细算法。在程序依赖图上，程序语句被表示成节点，语句之间的依赖关系用图的边表示。

2 克隆代码的预处理算法

本文提出的预处理算法由两个步骤组成。首先，

消除两种误检：插入或删除相同结构的语句和可交换的语句顺序。其次，在消除缺陷后的克隆代码中采用本文提出的识别方法识别可重构的克隆代码。

2.1 算法描述

本文提出的结合自适应K-最近邻聚类和程序依赖图的克隆代码预处理方法流程如图3所示。

该方法首先使用工具CPBugdetector[3]分别检测待测系统中的克隆代码和拷贝–粘贴相关的缺陷，使用基于代价–收益的评估方法对未检测出缺陷的克隆代码片段，识别适于重构的克隆代码；对检测出有缺陷的克隆代码，使用本文提出的结合A-KNN聚类和程序依赖图的方法消除这些误检。然后，再对拷贝–粘贴相关缺陷进行检测，加入人工参与部分，由人工来确认缺陷和对缺陷进行修正，将非误检又不属于疑似缺陷的克隆代码作为下一步的输入，识别适于重构的克隆代码。最后，对适于重构的克隆代码采用相应的重构方法进行重构，目的是消除程序中的克隆代码。

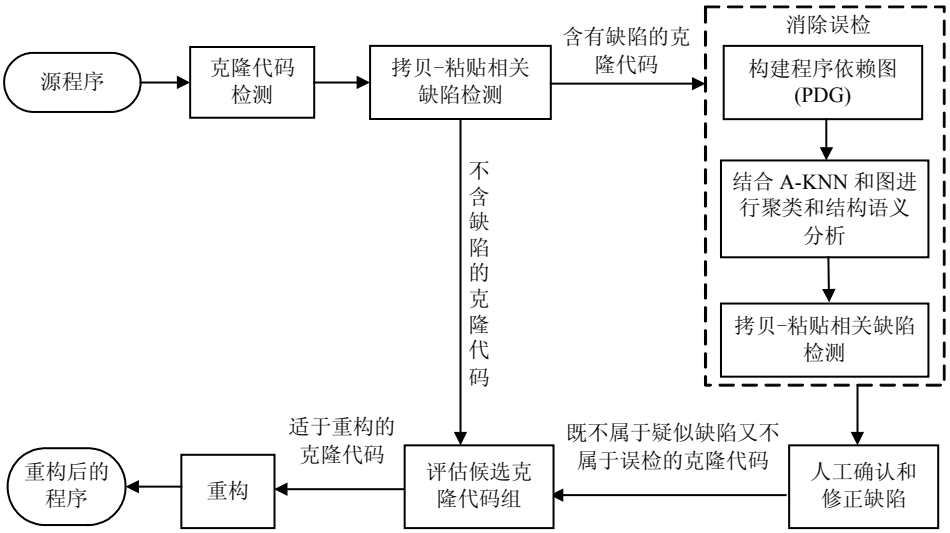

图3　可重构克隆代码的预处理算法框图

2.1.1 自适应K-最近聚类与程序依赖图相结合的误检消除算法

误检消除算法将自适应K-最近邻聚类与程序依赖图相结合，目前该方法主要用来分析C语言编写的程序，算法也可经修改后应用于其他类型的编程语言。算法描述如下。

算法：PruneFalsePositives

输入：含有缺陷的克隆代码的程序依赖图P

输出：消除误检后的克隆代码

Begin

构建实体属性矩阵**creatmatrix**(*p*)；

计算相似度矩阵**similarity**(*p*)；

分类结果添加到结果集合stackre_1和stackre_2中；

while(stackre_1&&stackre_2不空){

比较两个栈顶元素node1和node2的行号；

if(match){

stackre_1.pop(node1)；stackre_2.pop(node2);}

else{将未匹配节点放入集合collect中；}}\\endwhile

for(collect的节点){

if(一行未匹配){

将collect中节点指针指回其PDG,寻找与其哈希值相同的节点，判断它们之间是否有依赖;　}

else\\两行未匹配{

if(未匹配节点的hashvalue相等){

collect中节点指针分别指回其PDG，判断它们之间是否有依赖;　}}}

End

算法由3个步骤组成：1) 进行数据的收集；2) 聚类；3) 消除误检。经过步骤2)聚类之后，代码中关系最密切的实体(语句的行号)被聚集到一个集合中，然后比较每个聚类集合中的行号，确定发生误检的位置。

如果一个聚类集合中的两条语句与另一个聚类因以下两种原因引起的误检：“插入或删除结构相同的语句”和“可互换语句顺序”。经过步骤2)聚类，关系最密类集合中的两条语句未匹配，则比较它们的哈希值。如果哈希值相等(哈希值相同的语句是结

构相同的语句)，即为“语句可交换顺序”而引起的误检，然后在程序依赖图上执行结构语义分析，判断两条语句是否可交换，如果两条语句之间存在着数据依赖，此时不可交换，反之可交换。

如果一个集合中的语句未在另一个集合中找到可与之匹配的语句，存在两种可能：一种是疑似缺陷，另一种是因插入或删除相同结构的语句而引起的误检。此时，在程序依赖图上寻找与未找到匹配语句的哈希值相同的语句节点，如果找到，即为因“插入或删除相同结构的语句”而引起的误检，然后在程序依赖图上进行结构语义分析。如果在程序依赖图上未找到与未匹配语句的哈希值相同的语句节点，则为疑似缺陷。下面以图2程序为例，对算法进行详述。图2所示程序的误检是由于两个程序片段中的语句15和语句16的顺序发生颠倒产生的，表1所示为片段1的实体–属性矩阵。程序的实体–属性矩阵作为算法的输入，程序语句为算法的实体，控制变量和数据变量为属性。如表1所示，矩阵的行是该变量所在的语句行号。

步骤如下：首先，待分析程序的实体–属性矩阵作为算法的输入，用以构建程序的相似度矩阵，结果如表2所示。然后，算法处理声明语句节点，得到每个实体的类标签。最后，执行聚类算法，对语句进行层次聚类，得到聚类结果。

片段1:{12,15}{13,14}{16,19,17,18}，片段2:{12,16}{13,14}{15,19,17,18}。

表1 示例程序(图2)中片段1程序的实体-属性矩阵

实体	属性										控制属性
	error	out_putf	lastdirent	buf.previous	buf.error	file->f_pos	lastdirent->d_off	put_usser	count	buf.count	if
12	0	0	2	0	0	0	0	0	0	0	0
13	1	0	0	0	0	0	0	0	0	0	1
14	0	2	0	0	0	0	0	0	0	0	1
15	0	0	2	2	0	0	0	0	0	0	0
16	2	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	1
18	0	0	0	0	0	2	2	2	0	0	1
19	2	0	0	0	0	0	0	0	2	2	1

表2 示例程序(图2)中片段1程序的相似度矩阵

行号	12	13	14	15	16	17	18	19
12	1	0	0	0.5	0	0	0	0
13	0	1	0.214	0	0	0	0	0
14	0	0.214	1	0	0	0	0	0
15	0.5	0	0	1	0	0	0	0
16	0	0	0	0	1	0	0	0.23
17	0	0	0	0	0	1	0.11	0.11
18	0	0	0	0	0	0.11	1	0.06
19	0	0	0	0	0	0.11	0.06	1

经过聚类后，首先比较3个聚类集合中的行号，找出不匹配的行号，在本例中，语句15和16不匹配。对为匹配的语句，先比较它们的哈希值，如果哈希值相等，则对相应的程序依赖图进行结构语义分析，目的是确定两个语句间是否存在数据依赖，片段1程序的程序依赖图如图4所示。语句15和16间不存在数据依赖，因此可以交换语句15和16的顺序。

2.1.2 基于代价-收益分析的评估方法识别可重构的克隆代码

过程提取作为重构方法之一，可减少系统中的克隆代码，但经过过程提取，函数间的耦合度增加了，因而增大了软件维护的代价，而过程提取引起的代码规模的减少是其带来的收益。文献[6]通过代码间的耦合度计算来度量代码迁移的难易程度，然后，根据难易程度对语句进行排序，识别并提取可重构的克隆代码，认为使用外部定义的变量(被引用和被赋值)越少，代码的迁移难度越低。文献[12]在过长方法中识别可以被提取的片段，将代价和收益进行相除，根据比率对候选的片段进行排序，比率最高的片段被认为是适于提取的。本文在

文献[6,12]的基础上，提出了基于代价–收益评估的方法，识别可重构的克隆代码。该算法将代码行的减少视为重构的收益，而重构的代价是被提取的函数和周围代码之间的传递参数的数量。

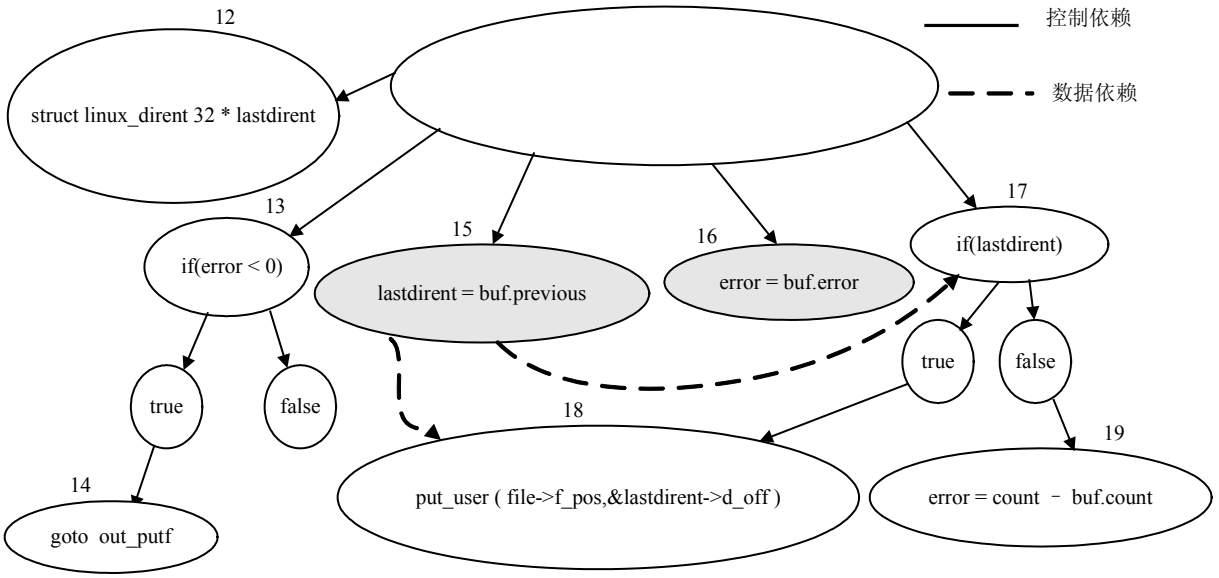

图4　示例程序(图2)中片段1的程序依赖图

1) 收益分析

克隆代码组F包括克隆代码片段$f_1, f_2, \cdots, f_m$，提取克隆代码组F的收益可以表示为：

$$\text{Benefit}(F)=m(|\text{cf}|-1) \tag{1}$$

式中，|cf|是F中每个片段内能被提取代码语句数量。在某些克隆代码中，存在一定数量的非克隆代码，但由于语句间的依赖关系，这些非克隆语句不能被移出。在已有的方法中，由于过程提取产生了一个过程调用，因此，实际语句的减少量为|cf|−1。

举例：图5所示程序含有克隆代码的片段的行数减少了5行(其中加粗显示的是克隆代码语句)，但由于过程调用，收益是8。

行号	片段 1	行号	片段 2
…		…	
7	if(get_user(kernel_mask, user_mask_ptr)) {	2	mm_segment_t old_fs;
8	return -EFAULT; }	3	int ret;
9	old_fs = get_fs();	4	old_fs = get_fs();
10	set_fs(KERNEL_DS);	5	set_fs(KERNEL_DS);
11	ret = sys_sched_setaffinity();	6	ret = sys_sched_setaffinity();
12	set_fs(old_fs);	7	set_fs(old_fs);
13	if(ret > 0)	8	ret = sizeof(compat_ulong_t);
14	ret = sizeof(compat_ulong_t);		

图5　Linux 2.6.6中检测出的克隆代码

2) 代价分析

耦合度表示过程提取的代价。克隆代码与它周围代码的耦合度越低，则克隆代码越易被移动[12]。在常见的7种耦合关系中，本文主要计算数据耦合，原方法和新方法(过程提取的结果)之间的耦合度通过计算新方法的参数来计算，有：

$$\text{Coupling}(F)=\sum_{i=1}^{m}(|P(i)_{\text{in}}|+|P(i)_{\text{out}}|) \tag{2}$$

式中，$|P(i)_{\text{in}}|$和$|P(i)_{\text{out}}|$分别是新方法输入参数和输出参数的数量。

举例：图5中所示的片段，old_fs和ret是输入参数，ret为输出参数，因此，此例中过程提取的代价为6。

综上所述，代价-收益分析的评估方法为：

$$R(F)=\begin{cases}\dfrac{\text{Benefit}(F)}{\text{Coupling}(F)}= \\ \dfrac{m(|\text{cf}|-1)}{\sum_{i=1}^{m}(|P(i)_{\text{in}}|+|P(i)_{\text{out}}|)} & \text{Coupling}(F)>0 \\ \text{Benefit}(F)=m(|\text{cf}|-1) & \text{Coupling}(F)=0\end{cases} \tag{3}$$

如果$R(F)\geqslant 1$，那么可提取该克隆代码组，反之不然。此外，一些仅由声明语句组成的克隆代码，不适于提取。式(3)中的变量含义如下：

$$P(i)_{\text{in}} = V_b \cap (V_w \cup V_r)$$

$$P(i)_{\text{out}} = V_a \cap V_w$$

式中，V_a、V_b、V_w、V_r都表示变量，V_a出现在被提取片段之后，V_b出现在被提取片段之前，V_w在被提取片段外部定义，在片段中被修改，V_r在被提取片段外部定义，在片段中被使用但未被修改。

表3 被测开源软件中克隆代码的检测结果

测试程序	C文件数量	代码行	克隆代码行	检测出的克隆代码组
linux 2.6.6/kernel	47	30 629	1 887	140
linux 2.6.6/arch	2 363	725 681	133 598	55 34
linux 2.6.6/net	536	333 741	61 585	2 543
linux/sound/drivers	24	12 380	493	75
unix/make 3.82	38	33 864	876	68
httpd 2.2.2/server	44	36 926	2 005	121
devecot 2.0.8	705	233 113	39 544	2 838
iptables 1.4.10	104	32 497	5 905	288
nginx 0.8.15	150	101 226	8 732	557

举例：图5中所示的克隆代码片段，由于过程提取的收益是8，而代价为6，$R(F)\geqslant 1$，因此表3中的克隆代码可被提取。

3 实验结果与分析

本文选取了9个由C语言编写的开源程序作为实验对象。首先采用克隆代码检测工具CPBugdetector[3]对这9个开源程序进行克隆代码检测及其相关缺陷的检测，克隆代码检测结果如表3所示。然后，使用本文提出的方法对测试程序进行误检消除，误检消除的实验结果如表4所示。

表4中记录了误检消除前后，分别针对检测出的“忘记修改某标识符”和“错误修改某标识符”的缺陷，检测工具报告的bug数量。表4的最后4列数据记录了本文提出的算法能够消除的误检的成因及缺陷类型。

表4 不同类型分类结果比较

测试程序	忘记修改某标识符		错误修改某标识符		本文算法可以处理的误检			
					可交换顺序的语句		插入或删除相同结构的语句	
	误检消除之前报告的bug	误检消除之后报告的bug	误检消除之前报告的bug	误检消除之后报告的bug	忘记修改某标识符	错误修改某标识符	忘记修改某标识符	错误修改某标识符
linux 2.6.6/kernel	6	6	4	4	0	0	0	0
linux 2.6.6/arch	38	38	214	179	0	6	0	29
linux 2.6.6/net	23	23	121	85	0	14	0	22
linux/sound/drivers	1	1	3	0	0	0	0	0
unix/make 3.82	0	0	3	3	0	0	0	0
httpd 2.2.2/server	1	1	5	5	0	0	0	0
devecot 2.0.8	84	83	155	143	0	1	1	11
iptables 1.4.10	5	5	4	4	0	0	0	0
Nginx 0.8.15	10	10	74	54	0	0	0	20

表5 识别可重构克隆代码

测试程序	检测出来的克隆代码组n_1	适于重构的克隆代码组(消除误检之前)	适于重构的克隆代码组n_2(消除误检之后)	$\frac{n_2}{n_1}/\%$
inux 2.6.6/arch	5 534	4 554	4 537	82.6
Linux 2.6.6/sound/drivers	75	61	61	81.3
Unix/make 3.82	68	57	57	83.8
Httpd 2.2.2/server	121	81	81	66.9

结合表3和表4进行分析，被测程序的规模越大，检测出来的bug越多，同时能被消除的误检也越多。最后，采用基于代价–收益分析的方法对消除误检之后的克隆代码识别适于重构的克隆代码，实验结果如表5所示。误检消除之后增加了适于重构的克隆代码的数量，同时，检测出来的克隆代码约70%是适于提取的。

参 考 文 献

[1] CAI D, KIM M. An empirical study of long-lived code clones[J]. Fundamental Approaches to Software Engineering Lecture Notes in Computer Science, 2011, 6603: 432-446.

[2] 于冬琦, 彭鑫, 赵文耘. 使用抽象语法树和静态分析的重复代码自动重构方法[J]. 小型微型计算机 2009, 30(9): 1752-1760.
YU Dong-qi, PENG Xin, ZHAO Wen-yun. Automatic refactoring method of cloned code using abstract syntax tree

and static analyis[J]. Journal of Chinese Computer Systems, 2009, 30(9): 1752-1760.

[3] 苏小红, 马培军, 王倩, 等. C克隆代码缺陷检测工具: 中国, [CPBugdetector] V1.0[CP/DK]. 2010.
SU Xiao-hong, MA Pei-jun, WANG Qian, et al. A defects detection tool for C clones, China, [CPBugdetector] V1.0[CP/DK]. 2010.

[4] KIM M, BERG L, LAU T, et al. An ethnographic study of copy and paste programming practices in oopl[C]// International Symposium on Empirical Software Engineering. Washington DC, USA: IEEE, 2004: 83-92.

[5] CODE N, KOS R. Frequency and risks of changes of clones[C]//33rd Internaitonal Conference on Software Engineering. Hawaii, USA: IEEE, 2011: 311-320.

[6] HIGO Y, KUSU S, INOUE K. Identifying refactoring opportunities for removing code clones with a metrics-based approach[M]. Hong Kong, China: CreateSpace Independent Publising Platform, 2011: 1-26.

[7] CHOU A, YANG J, CHELF B, et al. An empirical study of operating systems error[J]. SIGOPS Operating Systems Review, 2001, 35(1): 73-88.

[8] LI Z, LU S, MYAGMAR S, et al. CP-miner: Finding copy-paste and related bugs in large-scale software code[J]. IEEE Transactions on Software Engineering, 2006, 32(3): 176-192.

[9] 王倩. 基于序列挖掘的C克隆代码及相关软件缺陷的检测[D]. 哈尔滨: 哈尔滨工业大学, 2009.
WANG Qian. Detection of clones and related software defects of C programs via sequential pattern mining[D]. Harbin: Harbin Institute of Technology, 2009.

[10] MA Pei-jun, BIAN Yi-xin. A clustering method for pruning false positive of clone code detection[C]// International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC 2013). Shenyang, China: IEEE, 2013: 1917-1920.

[11] KAPSER C, GODFREY M W. “Cloning considered harmful” considered harmful[C]//The 13th Working Conference on Reverse Engineering. Washington DC: USA: IEEE, 2006: 19-28.

[12] YANG L, LIU HUI, NIU Z. Identifying fragments to be extracted from long methods[C]//Asia-Pacific Software Engineering Conference. Washington DC, USA, IEEE, 2009: 43-49.

[13] BIAN Yi-xin, SU Xiao-hong, MA Pei-jun. Identifying accurate refactoring opportunities using metrics[C]// International Conference on Soft Computing Techniques and Techniques and Engineering Application. Shenyang, China: IEEE, 2013: 141-146.

[14] ALKHALID A, ALSHAYEB M, MAHMOUD S. Software refactoring at the function level using new adaptive K-nearest neighbor algorithm[J]. Advances in Engineering Software, 2010, 41(10-11): 1160-1178.

[15] FERRANTE J, OTTENSTEIN K J, WARREN J D. The program dependence graph and its use in optimization[J]. ACM Transaction on Program Language System, 1987, 9(3): 319-349.

编 辑 漆 蓉