第42卷 第6A期
2015年6月

计 算 机 科 学
Computer Science

Vol. 42 No. 6A
June 2015

上下文不一致性缺陷的检测

王 伟 刘 渊 张春瑞 文 平 谢家俊
(中国工程物理研究院计算机应用研究所 绵阳 621900)

摘 要 为了检测在软件开发过程中由拷贝-粘贴操作引起的上下文不一致性缺陷,在基于频繁子序列挖掘算法的克隆代码检测模型基础上,改进上下文不一致性缺陷检测过滤规则,对上下文结构类型不一致性和上下文条件谓词不一致性两种缺陷进行了检测。为了识别具有相同语义但不同语法结构(即语法树表示)的表达式,还增加了对生成的表达式语法树的标准化处理。开源代码上的实验结果表明,该模型对拷贝-粘贴操作引起的上下文不一致性缺陷具有较低的误检率,不存在漏检,尤其适用于安全攸关的软件系统。

关键词 序列挖掘,克隆代码,上下文不一致性缺陷,软件缺陷检测
中图法分类号 TP311 文献标识码 A

Detection of Context-based Inconsistencies Bugs

WANG Wei LIU Yuan ZHANG Chun-rui WEN Ping XIE Jia-jun
(Institute of Computer Application, China Academy of Engineering Physics, Mianyang 621900, China)

Abstract In order to detect context-based inconsistencies bugs induced by copy-paste in the development of software, based on the model of clone code detection via sequential pattern mining algorithm, filter rules of context-based inconsistencies were improved. Both context constructs inconsistency bugs and context conditional predicates inconsistency bugs were detected. To recognize semantically equivalent with different syntactic structure(i. e. syntax-tree), the standardization of expressions syntax-tree was added. The experimental results on the open source codes show that the model has low false-positive rate and 0% false-negative rate. It is especially suitable for safety-critical software.

Keywords Sequential pattern mining, Clone code, Context-based inconsistencies bugs, Software bug detection

在软件开发过程中,由于拷贝-粘贴操作的广泛使用,使得大型软件中克隆代码比例很高,有统计数据表明大型软件中约7%~23%的代码是重复的。拷贝-粘贴后常因修改引入缺陷,随着软件规模的不断增大,此类软件缺陷也不断增多,严重影响了软件的可靠性,对于安全攸关(Safety-Critical)的软件来说,该问题尤为突出。

克隆代码(Clone Code),通常是指软件中存在的完全相同或经过少量修改的相似代码片段。目前,已有很多软件缺陷检测工具,但通常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。在常用的克隆代码检测方法中,基于抽象语法树的方法[1]通过建立和搜索语法树来分析程序的语法结构,如JPLag[2],Moss[3],sif[4],DECKARD[5]等;基于程序语义的方法[6,7],通过建立和搜索程序依赖图来分析程序的语法结构,这两种方法都能识别语义等价、语法结构改变的克隆代码,但时间复杂度高,因而不适用于大型软件。基于字符串的方法(如DUPLOC[8],Diff[9],Baker[11],Dup[12]等)和基于token流的方法(如Dotplots[13],CCFinder[14],Hamid[15],YAP[16]等)虽然时间和空间复杂度低,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,而且相关工具如CCFinder等仅能检测克隆代码,不能识别由克隆代码引起的软件缺陷。

虽然克隆代码检测的工具层出不穷,但是在此结果上进行相关软件缺陷检测的技术却寥寥无几。克隆代码引起的缺陷主要包括标识符重命名不一致性缺陷和上下文不一致性缺陷。本文重点研究上下文不一致性缺陷的检测。目前L Jiang等在对克隆代码的不一致性深入分析的基础上,提出了上下文不一致性的概念和上下文不一致性缺陷的两种类型,进而提出了基于抽象语法树的特征向量聚类进行克隆代码及上下文不一致性缺陷检测[17]的方法。

与文献[17]不同的是,本文采用基于token流的方法检测克隆代码。为了降低漏检和误检,基于频繁子序列挖掘的克隆代码检测模型,改进了文献[17]中的上下文不一致性缺陷检测过滤规则,同时在条件谓词匹配中对表达式语法树进行了标准化,减少了"表达式语义相同但语法结构不同"的误检情况。实验表明,该方法能够检测出更多类型的上下文结构类型不一致性缺陷,并且由于基于token流的方法对于克隆代码的检测效率较高,因而具有较好的缺陷检测效率。

1 基于序列挖掘的C克隆代码及上下文不一致性缺陷检测模型

克隆代码检测是为了解决与克隆代码相关的问题。本文关注的是如何在克隆代码检测的基础上,检测其相关的上下

本文受中物院科学技术发展基金(2012A0403021)资助。

王 伟(1986—),男,工程师,主要研究领域为软件安全检测、信息安全等;刘 渊(1974—),男,高级工程师,主要研究领域为信息安全等;张春瑞(1980—),男,高级工程师,主要研究领域为协议识别等。

文不一致性软件缺陷。本文基于频繁子序列挖掘的克隆代码检测模型[18]，提出了基于序列挖掘的C克隆代码及相关上下文不一致性缺陷检测模型。该模型在克隆代码检测后，根据其生成的克隆代码集合以及克隆代码相关源程序信息，进行了上下文不一致性缺陷的检测，其模型如图1所示。

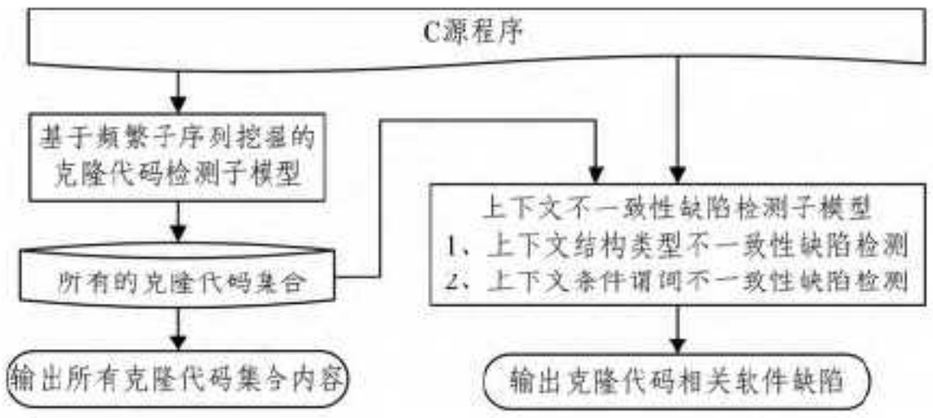

图1 基于序列挖掘的C克隆代码及上下文不一致性缺陷检测模型

2 基于序列挖掘的C克隆代码及上下文不一致性缺陷检测

2.1 克隆代码的上下文不一致性

根据克隆代码的产生原因和特点，我们认为相似的代码应该具有相似的功能并且在相似的上下文中使用。基于这种假设，可以推断出具有不同上下文的克隆代码有可能存在软件缺陷。“上下文”的定义是克隆代码上下文不一致性判断和相关缺陷检测的基础。

定义1(上下文)[17] 一个代码片段F的上下文是包含F的最内层的控制结构。例如，在C语言中，if语句，switch语句，for语句，while语句以及函数定义等，就是这种控制结构。在Java语言中，类定义也是这种结构。

算法1[17]基于程序的抽象语法树中间表示形式，描述了如何查找克隆代码上下文。对于给定的包含克隆代码的程序语法分析树，自底向上对树进行搜索，从而找出包含克隆代码的最内层树节点，这个节点就是一个上下文节点，也就是包含克隆代码的最内层的控制流结构。

算法1 CONTEXT(T: tree, F: clone): node
输入：克隆代码F与所属代码的语法树T
输出：克隆代码的上下文在语法树T中对应的节点C_R
1. 查找T中的最小子树T_F，满足T_F完全包含F；
2. 设R为T_F的根节点；
3. 找到R的最年轻的上下文祖先节点C_R；
4. 返回C_R；

本文用程序的token流而非抽象语法树，作为代码的中间表示，进行克隆代码检测，结合上下文的定义，本文给出了基于token流的克隆代码上下文查找算法，如算法2所示。

算法2 FindContext(CP-SEG, program) cntxtTknLine
输入：克隆代码信息CP-SEG，克隆代码对应C源代码字符串program
输出：克隆代码对应上下文的token串cntxtTknLine
1. 词法分析program，生成按行存储的token流srcTknLines；
2. 获取CP-SEG中的克隆代码位置信息cpLineInfo；
3. 根据cpLineInfo定位出srcTknLines中对应的克隆代码token行cloneTknLine；
4. 获取cloneTknLine中第一个节点记录的上下文行数信息cntxtLineInfo；
5. 根据cntxtLineInfo定位出TknLines中对应的上下文token行cntxtTknLine；
6. 返回cntxtTknLine；

文献[17]中克隆代码的上下文是指包含克隆代码的最内层的控制流结构，是对整个克隆代码段寻找上下文，存在漏检的情况(例如后面图5中的缺陷)，因此本文改进为对克隆代码的每一行寻找上下文并进行缺陷检测。算法首先对每个基本语句行的token串进行连接和处理，将程序的每行语句转化成一个数字序列项，这时输出的克隆代码信息是基于行号表示的。一般在词法分析阶段产生的token流并不包含程序的结构信息，但为了方便查找基于token流的克隆代码的上下文，本文在词法分析的后期，对token流进行换行标准化，使其按行存储，并为每个token记录了其所在语句对应的上下文token串的行号。

定义2(上下文不一致性类型1)[17] 对于给定的克隆代码对F_1和F_2，以及它们分别对应的上下文C_1和C_2，如果C_1和C_2的控制语句类别不同(用$KIND(C_1)\neq KIND(C_2)$表示)，则称F_1和F_2具有上下文不一致性类型1。用$I_1(F_1,F_2)$表示这样的不一致性，使得如果$KIND(C_1)\neq KIND(C_2)$，则$I_1(F_1,F_2)=1$；相反如果$KIND(C_1)=KIND(C_2)$，则$I_1(F_1,F_2)=0$。将这个定义提升到一个克隆代码集合，对于给定的克隆代码集合G，如果存在一个唯一的等价类划分$G/I_1=\{g_1,g_2,\cdots,g_k\}$，满足$k>1$，且

1) $\forall i(\forall C,C'\in g_i, I_1(C,C')=0)$；

2) $\forall i\neq j(\forall C\in g_i, \forall C''\in g_j, I_1(C,C'')=1)$。

则称G拥有上下文不一致性类型1，并且用$I_1(G)=k$表示G的这种不一致性。

图2[17]展示了具有上下文不一致性类型1的一个克隆代码对。

File: org.elipse.debug.ui/ui/org/eclipse/debug/ui/memory/AbstractTableRendering.java

```
Code1:
3557:int colCnt=fTableViewer.getTable().getColumnCount();
3558:TableItem item=fTableViewer.getTable().getItem(0);
3559:for(int i=0;i<colCnt;i++)
3560:{
3561:Point start=new Point(item.getBounds(i).x,…
3562:start=fTableViewer.getTable().toDisplay(start);
……
3565:if(start.x < point.x && end.x > point.x)
3566:return;
3567:}
Code2:
2697:TableItem item=null;
2698:for(int i=0;i<fTableViewer.getTable().getItemCount(),i++)
    item=…
2705:if(item != null)
2706:{
2707:  for(int i=0;i<colCnt;i++)
2708:  {
2709:    Point start=new Point(item.getBounds(i).x,…
2710:    start=fTableViewer.getTable().toDisplay(start);
      ……
2713:    if(start.x < point.x && end.x > point.x)
2714:      return i;
2715:  }
2716:  }
```

图2 上下文不一致性类型1示例

第 3559—3567 行和第 2707—2715 行为一个克隆代码对,"Code1"的上下文是一个函数定义,同时"Code2"的上下文是一个 if 语句。事实上,Code1 的第 3558 行被确认为一个上下文不一致性类型 1 缺陷,编程人员在此忽略了 Code2 中第 2705 行对 item 不等于 null 的条件检查。

定义 3(上下文不一致性类型 2)[17] 对于给定的克隆代码对 F_1 和 F_2,与它们分别对应的上下文中的控制结构条件谓词 P_1 和 P_2,如果 P_1 和 P_2 对应的语法树不能完全匹配,则称 F_1 和 F_2 具有上下文不一致性类型 2。用 $I_2(F_1,F_2)$ 表示这样的不一致性,使得当 P_1 和 P_2 不匹配时,$I_2(F_1,F_2)=1$,反之 $I_2(F_1,F_2)=0$。如果 F_1 和 F_2 没有对应的条件谓词,则 $I_2(F_1,F_2)=0$。将这一定义提升到一个克隆代码集合,对于一个给定的克隆代码集合 G,如果存在一个唯一的划分 $G/I_2=\{g_0,g_1,g_2,\cdots,g_k\}$,满足 $k>1$ 且

1) g_0 完全包含那些上下文没有谓词的克隆代码;

2) $\forall i\neq 0(\forall C,C'\in g_i,I_2(C,C')=0)$;

3) $\forall i\neq j\neq 0(\forall C\in g_i,\forall C''\in g_j,I_2(C,C')=1)$。

则称 G 具有上下文不一致性类型 2,并用 $I_2(G)=k$ 表示 G 的这种不一致性。

图 3[17] 中一对代码片段不具有上下文不一致性类型 1,它们的上下文都是 if 语句。但是它们具有上下文不一致性类型 2,因为它们的 if 语句条件分别调用了不同的函数。

File:linux-2.6.19/drivers/scsi/arm/eesox.c

Code3:

```
407:if(length>=9 && strncmp(buffer,"EESOXSCSI",9) ==0){
408:    buffer +=9;
409:    length -=9;
410:
411:    if(length >=5 && strncmp(buffer,"term",5) ==0) {
        ......
418:    } else
419:        ret=-EINVAL;
420:} else
421:    ret=-EINVAL;
```

File:linux-2.6.19/drivers/scsi/arm/cumana-2.c

Code4:

```
322:if(length>=11 && strcmp(buffer,"CUMANASCSI2") ==0)
    {
323:    buffer+=11;
324:    length-=11;
325:
326:    if(length >=5 && strncmp(buffer,"term",5) ==0) {
        ......
333:    } else
334:        ret=-EINVAL;
335:} else
336:    ret=-EINVAL;
```

图 3 上下文不一致性类型 2 示例

具有上下文不一致性类型 2 的克隆代码可能因不同的条件谓词控制,而执行不同的控制流路径,从而具有不同的行为。这种不一致性违反了关于相似代码在相似条件下具有相似行为的假设,因此可能存在缺陷。

本文将定义 2 中的"上下文不一致性类型 1"称为"上下文结构类别不一致性";将定义 3 中的"上下文不一致性类型 2"称为"上下文条件谓词不一致性"。

在 C 语言中,作为控制结构的语句有 for 语句、while 语句、do-while 语句、switch 语句、if 语句和函数定义语句。我们将循环结构语句 for、while 和 do-while 语句统称为 *loop* 语句,将选择结构语句 switch 和 if 语句统称为 *select* 语句,函数定义语句用 *fundef* 表示,从而上下文的结构类型分为 *loop*、*select* 和 *fundef* 3 种。对于一个克隆代码对 $\langle F_1,F_2\rangle$ 对应上下文 C_1 和 C_2,共有 6 种上下文组合类型,下面给出了克隆代码对的不一致子类型的定义。

定义 4(克隆代码对的不一致子类型)[17] 对于给定的克隆代码对 F_1 和 F_2,以及它们的上下文 C_1 和 C_2,通过 C_1 和 C_2 的类型组合,给出克隆代码对上下文子类型定义如下:

1) 子类型 1:$KIND(C_1)=fundef \wedge KIND(C_2)=fundef$;

2) 子类型 2:$KIND(C_1)=fundef \wedge KIND(C_2)=loop$;

3) 子类型 3:$KIND(C_1)=loop \wedge KIND(C_2)=select$;

4) 子类型 4:$KIND(C_1)=loop \wedge KIND(C_2)=loop$;

5) 子类型 5:$KIND(C_1)=select \wedge KIND(C_2)=select$;

6) 子类型 6:$KIND(C_1)=select \wedge KIND(C_2)=fundef$。

2.2 上下文不一致性缺陷检测

2.2.1 克隆代码上下文不一致性缺陷检测算法

本文的克隆代码上下文不一致性缺陷检测过程如图 4 所示。

```
CntxtInconsisDetect::operator()
输入:源文件 token 行转化表示类 filesToTokens,所有克隆代码集合
     的集合 CP-Seg-Array,头文件信息集合 headerFilesArry
输出:检测出的上下文不一致性缺陷集合
Begin
  foreach 克隆代码片段对⟨F1,F2⟩ in CP-Seg-Array
    词法分析 F1 和 F2 对应的源程序,生成 F1 和 F2 的 token 流按行
    存储;
    应用图 3 中的算法,找出 F1 和 F2 分别对应的上下文 C1 和 C2
    的 token 行;
    应用改进的过滤规则进行判断:
      if 存在疑似的上下文结构类型不一致性缺陷 then
        加入"上下文结构类型不一致性"缺陷列表;
      else
        进行条件谓词匹配;
        if 存在疑似的上下文条件谓词不一致性缺陷 then
          加入"错误修改某标识符"缺陷列表;
        endif
      endif
    endfof
    输出上下文不一致性缺陷信息;
end
```

图 4 克隆代码上下文不一致性缺陷检测算法伪代码

2.2.2 改进的上下文不一致性缺陷过滤规则

本文根据克隆代码对的上下文子类型的定义,参考文献[17]的过滤规则,提出了判断上下文不一致性是否可能存在缺陷的改进过滤规则,如表 1 所列。

文献[17]提出的过滤规则,将子类型 3(当上下文的类型为 *select*,而该上下文的上下文的类型不是 *loop* 的情况)作为非软件缺陷过滤掉。本文认为将这种情况过滤掉是不合理的,在这种情况下可能存在软件缺陷。

表 1　上下文不一致性过滤规则

子类型	具体形式	上下文不一致性	可能存在的缺陷
子类型 1	$KIND(C_1)=fundef \wedge KIND(C_2)=fundef$	无	无
子类型 2	$KIND(C_1)=fundef \wedge KIND(C_2)=loop$	上下文结构类型不一致性	增加或去掉循环结构导致的误检大多为开发人员有意为之，因此这种情形是缺陷的可能性极小
子类型 3	$KIND(C_1)=loop \wedge KIND(C_2)=select$	上下文结构类型不一致性	如果 C_2 的上下文的上下文类型是 loop，则可能是边界条件判断冗余或缺失；否则，可能是错误的循环控制范围
子类型 4 子类型 5	$KIND(C_1)=loop \wedge KIND(C_2)=loop$ $KIND(C_1)=select \wedge KIND(C_2)=select$	可能是上下文条件谓词不一致性，需要进一步条件谓词的匹配	错误的函数调用或错误的条件判断表达式
子类型 6	$KIND(C_1)=select \wedge KIND(C_2)=fundef$	上下文结构类型不一致性	边界条件判断冗余或缺失

如图 5 所示，上边代码第 5 行克隆代码的上下文为 *select*，该 *select* 的上下文为 *loop*，下边代码的第 5 行克隆代码的上下文为 *loop*，这对克隆代码属于上下文子类型 3。可以看出下边代码很可能是缺少 if 而导致了边界条件判断缺失。

```
-CP 1- static void ttusb_dec_free_iso_urbs[struct ttusb_
       {
-CP 2-   int i;
-CP 3-   dprintk["s",__FUNCTION__];
-CP 4-   for [i=0;i<ISO_BUF_COUNT;i++]
         {
           if [dec->iso_urb[i]]
           {
-CP 5-       usb_free_urb[dec->iso_urb[i]];
           }
         }
         pci_free_consistent[NULL,ISO_FRAME_SIZE]
       }
-CP 1- static void ttusb_dec_exit_usb[struct ttusb_dec*
       {
-CP 2-   int i;
-CP 3-   dprintk ["s",__FUNCTION__];
       dec->iso_stream_count=0;
-CP 4-   for [i=0;i<ISO_BUF_COUNT;i++]
         {
-CP 5-     usb_unlink_urb[dec->iso_urb[i]];
         }
       ttusb_dec_free_iso_urbs[dec];
}
```

图 5　上下文不一致性子类型 3 示例 1

如图 6 所示，上边代码的 4861—4864 行和下边代码的 2386—2390 行是一对克隆代码，上边的 4863 和 4864 行的上下文是 4859 行的 if 语句，且并不知道上边代码的 4859 行的 if 语句的上下文是否为 *loop* 类型，而对应于下边的 2389 和 2390 行的上下文是 2386 行的 for 语句，这对克隆代码属于上下文子类型 3。可以看出上边代码的 for 循环语句相对于下边代码的 for 语句，很可能因缺少了花括号而导致了错误的控制范围。

File: linux-2.6.19/drivers/cdrom/sbpcd.c

Code7:

```
4859: if(cmd_type ===READ_M2)
4860: {
4861:   for(xa_count=0;xa_count<CD_XA_HEAD;xa_count++)
4862:     sprint(&msgbuf[xa_count*3],"%0.2X",…);
4863:   msgbuf[xa_count*3]=0;
4864:   msg(DBG_XA1,"xa_head: %s\n",msgbuf);
Code8:
2386: for(i=0;i<response_count;i++)
2387: {
2388:   sprint(&msgbuf[i*3],"%0.2X",…);
2389:   msgbuf[i*3]=0;
2390:   msg(DBG_SQ1,"cc_ReadSubQ: %s\n",msgbuf);
2391: }
```

图 6　上下文不一致性子类型 3 示例 2

图 5、图 6[17] 是两种可能存在缺陷的子类型 3 在文献[17]中被过滤掉的例子。因此，本文在子类型 3 中改进了文献[17]的过滤规则，消除了文献[17]漏检的情况。

2.2.3　上下文条件谓词的匹配

为了进一步匹配上下文的条件谓词，本文将循环结构和选择结构的条件表达式表示为抽象语法树的形式。除去条件运算符后，表达式中的各种操作符就只剩下一元、二元操作符，因此本文用二叉树来表示上述条件表达式。用语法树表示表达式有两个优点：一是语法树的各种遍历算法为本文的表达式语法树的建立和匹配提供了方便；二是在保持原表达式语义不变的基础上去除了表达式中的括号“(”和“)”，进一步提高了表达式语法树的遍历和匹配效率。

这里循环结构和选择结构的条件表达式可以是算术表达式、关系表达式或者逻辑表达式。本文采用算符优先分析原则为这 3 种表达式生成语法树。

为了识别具有相同语义但不同语法结构(即语法树表示)的表达式，本文还按照文献[19]给出的规则对生成的表达式语法树进行了如下标准化。

1) 算术表达式标准化

算术表达式定义为 t1@t2。t1，t2 是对象或算术表达式；@代表算术运算符，大多数程序的算术表达式中有“+”、“—”、“*”、“/”、“%” 5 种运算符，在表达式中可使用圆括号限定运算顺序。我们利用这些运算符的结合律、交换律和分配律等性质对算术表达式进行标准化。

算术表达式标准化的处理过程是按照顺序应用制定的 33 条算术表达式标准化规则，反复调用这些规则，直到每一条规则都不再适用于表达式为止。由于本文中所有表达式均是用语法分析子树表示，因此只需要应用适当树的遍历算法及一些基本转换操作就可以实现表达式的标准化工作。下面是几个算术表达式标准化的例子：

$-j/1+k;\Rightarrow k-j;$

$-(-j);\Rightarrow j;$

$h=i/k*m;\Rightarrow h=i*m/k;$

$i*(j+k)-(j-k);\Rightarrow i*j+i*k+k-j;$

2) 布尔表达式标准化

本文将各种逻辑表达式、关系表达式及用在控制语句中的条件表达式统称为布尔表达式。布尔表达式标准化的处理过程是按照顺序应用制定的 27 条布尔表达式标准化规则，重复使用这些规则，直到其中任何一条都不再适用为止。布尔表达式进行标准化时，首先分离出其中的算术表达式成分，调

用算术表达式标准化程序处理这些算术表达式，然后再应用布尔表达式标准化算法处理。下面是3个布尔表达式标准化的例子：

$m<5;\Rightarrow 5>m;$

$(i||k)\&\&j;\Rightarrow i\&\&j||k\&\&j;$

$!(k==n);\Rightarrow k!=n;$

$i==j||i!=j;\Rightarrow 1;$

标准化后完成对表达式语法树的匹配[20]，用[0,1]区间内的小数来表示两个语法树的相似程度，这个数被称为语法树的相似度 $similarity$。在对表达式建立语法树并经过标准化后，语义相同的表达式语法树一定是完全相同的，那么匹配度为1；而语义不同的表达式语法树一定是部分不同或完全不同，如果是部分不同，则匹配度为0和1之间的小数，如果完全不同，则匹配度为0。

本文进行上下文条件谓词语法树匹配是为了判定拷贝-粘贴操作引起的上下文不一致性缺陷，因为相似的代码通常实现相似的功能，并在相似的上下文中使用，基于这一假设，可以认为具有不同上下文的克隆代码有可能含有缺陷，进一步可以推断，克隆代码不同的上下文的相似程度越大，越可能存在缺陷。根据语法树匹配度的计算结果，可知当匹配度等于0或1时，都不太可能存在上下文条件谓词不一致性缺陷；而在区间(0,1)之间时，匹配度越大，越可能存在上下文条件谓词不一致性缺陷。因此，本文定义一个表达式匹配度阈值 $SimilarityThreshold(>0)$，当 $similarity \geqslant SimilarityThreshold$ 时，判定为疑似上下文条件谓词不一致性缺陷。本文采用文献[20]的表达式匹配算法，但在表达式树节点匹配过程中，为了进一步提高效率，遵循以下新的规则：

1)两个变量匹配时只需检查变量的类别，检查变量是属于数字型、字符型、数组型还是指针型等其他复杂结构变量，如果这些信息不匹配，则这两个变量匹配不成功，常量匹配时要检查值是否相等。

2)匹配两个函数调用节点时，如果两个函数都是C库函数，则检查两个函数名是否相同；否则，检查两个函数的参数个数是否相同。

3 实验结果与分析

将Linux2.6.6源代码中的sound\driver和drivers\media子模块作为实验代码，进行上下文不一致性缺陷的检测，表达式匹配相似度的阈值设为默认值。实验结果如表2、表3所列。

表2 上下文结构不一致性缺陷实验结果

源代码及行数	人工注入的缺陷个数 (#Insertbugs)	检测出的缺陷个数 (#Suspects)	确认的人工注入的缺陷个数 (#Bugs)	人工注入的缺陷的漏检率 (TP%)	人工注入缺陷的误检率 (FP%)	检测时间上限 (#Time(sec))
linux-2.6.6\sound\ driver(12380)	20	17*	20	0	17.6%	101
linux-2.6.6\drivers\ media(119164)	50	74*	50	0	32.4%	2830

表3 上下文条件谓词不一致性缺陷实验结果

源代码及行数	人工注入的缺陷个数 (#Insertbugs)	检测出的缺陷个数 (#Suspects)	确认的人工注入的缺陷个数 (#Bugs)	人工注入的缺陷的漏检率 (TP%)	人工注入缺陷的误检率 (FP%)	检测时间上限 (#Time(sec))
linux-2.6.6\sound\ driver(12380)	20	20	20	0	0%	90
linux-2.6.6\drivers\ media(119164)	40	45*	40	0	11.1%	3931

其中，误检率计算公式为：$FP\% = \frac{\#Suspects - \#Bugs}{\#Suspects} \times 100\%$，漏检率计算公式为：$TP\% = \frac{\#Insertbugs - \#Bugs}{\#Insertbugs} \times 100\%$。由于上下文结构不一致性和条件谓词不一致性两类缺陷是同时被检测的，但两类缺陷的检测实验是对源代码分别注入进行的，因此没有给出单类缺陷检测的精确运行时间，只给出了检测时间上限(#Time)。带"*"的缺陷中误检出的缺陷是由于注入缺陷的代码片段在其他的克隆代码组中出现，而不同的克隆代码行拥有相同的上下文，因此导致注入的该缺陷被重复检测出来，其实相当于在其他的克隆代码组中间接注入了该缺陷，或者是由于克隆代码组中有多个重复片段($\geqslant 3$)组合检测所致，因此不是真正的误检，而是间接注入的缺陷，从这个意义上说，误检率为0%。

图7中，克隆代码组24中的代码片段1和克隆代码组97中的代码片段2为同一个代码片段，实验中在克隆代码组24的代码片段1中注入了缺陷(error5)，灰色矩形部分为注入的缺陷控制结构，该缺陷被正确检测出来。然而在克隆代码组97中，由于和代码片段1构成克隆代码的片段2的上下文恰好是注入的缺陷控制结构，因为也被当成缺陷被检测出来，导致了误检。可以看出，克隆代码组97中的这个缺陷是由于同一个代码片段分散在不同的克隆代码组中而导致的，是间接注入的，因此不是真正的误检。

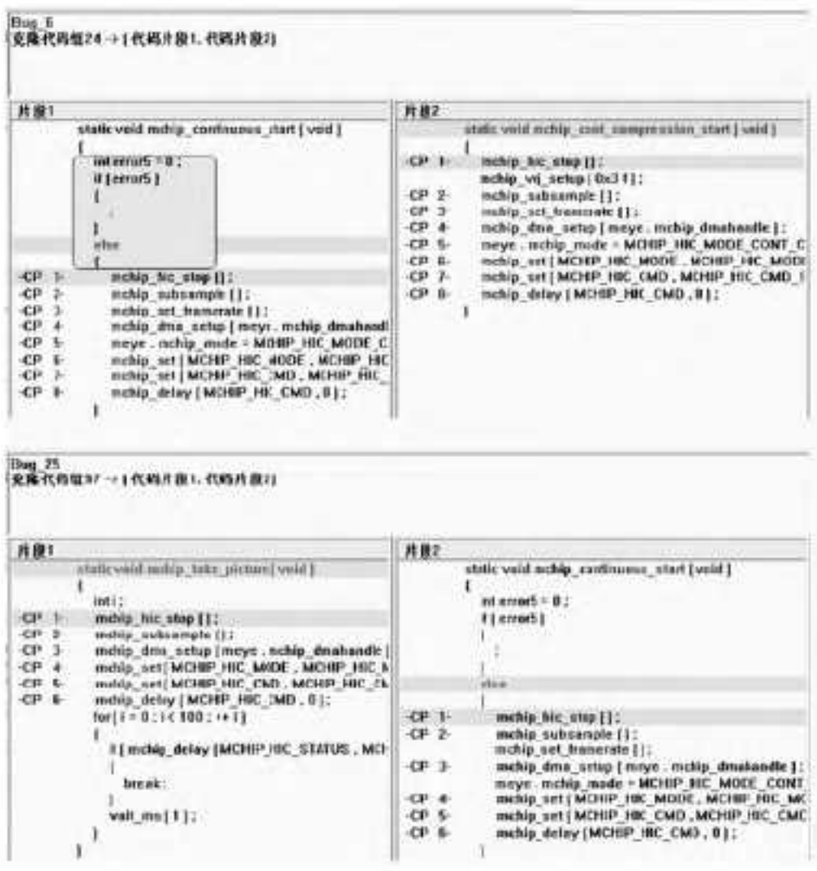

图7 上下文结构不一致性缺陷误检示例

对于安全关键的软件系统而言，一方面缺陷被遗漏到下一阶段将使修复缺陷的成本扩大到原来的5—10倍甚至无法修复，另一方面软件发生错误有可能造成巨大的经济损失，因此相对于降低误检而言，我们更关注的是降低漏检。从表2

和表 3 的实验结果可以看出，本文提出的检测模型对人工注入的缺陷具有较低的漏检率和误检率，尤其是漏检率全部为 0，说明该方法特别适用于对软件可靠性要求较高的安全关键的软件系统的缺陷检测，由于编程的灵活性和复杂性，检测到的疑似缺陷均需开发人员进一步确认是开发人员有意而为之（即误检）还是真正的缺陷。

结束语　本文在检测克隆代码的基础上进行上下文不一致性缺陷的检测，通过改进上下文不一致性缺陷检测的过滤规则和对建立的表达式抽象语法树进行标准化处理，能够在更广的范围上检测出上下文结构不一致性和上下文条件谓词不一致性两种缺陷。实验表明，该模型对这两种缺陷有较低的漏检率和误检率，能够有效地帮助开发人员尽早发现缺陷，提高软件质量。

参 考 文 献

[1] Baxter I D, Yahin A, Moura L, et al. Clone Detection Using Syntax Trees[C] // Proc. Int'l Conf. Software Maintenance. 1998: 368-377

[2] Prechelt L, Malpohl G, Philippsen M. Finding Plagiarisms among a Set of Programs with JPlag[J]. J. Universal Computer Science, 2002: 1016-1038

[3] Schleimer S, Wilkerson D S, Aiken A. Winnowing: Local Algorithms for Document Fingerprinting[C] // Proc. ACM SIGMOD Int'l Conf. Management of Data. 2003: 76-85

[4] Manber U. Finding Similar Files in a Large File System[C] // Proc. USENIX Winter 1994 Technical Conf. 1994: 1-10

[5] Jiang L, Misherghi G, Su Z, et al. Deckard: Scalable and accurate tree-based detection of code clones[C] // ICSE. 2007: 96-105

[6] Komondoor R, Horwitz S. Using Slicing to Identify Duplication in Source Code[C] // Proc. Eighth Int'l Symp, Static Analysis. 2001

[7] 李建忠，刘建宾，余楚迎. 基于过程蓝图的参数化重复代码检测技术研究[J]. 汕头大学学报：自然科学版，2007，22(1)：54-59

[8] Ducasse S, Rieger M, Demeyer S. A Language Independent Approach for Detecting Duplicated Code[C] // Proc. Int'l Conf. Software Maintenance. 1999: 109-118

[9] Van Rysselberghe F. Vingerafdrukken van code om duplicatie op te sporen[D]. Antwerper: Master Thesis at Universiteit Antwerpen, 2001-2002

[10] Baker B. A Program for Identifying Duplicated Code[C] // 24th Symposium on the Interface Proceedings of Computing Science and Statistics. 1992: 132-141

[11] Baker B. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenanc[C] // 25th Annual ACM Symposium on Theory of Computing. 1993: 75-96

[12] Baker B S. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance[J]. SIAM Journal on Computing, 1997, 26(5): 1343-1362

[13] Church K W, Helfman J I. Dotplot: A Program for Exploring Self-Similarity in Millions of Lines of Text and Code[J]. J. Computational and Graphical Statistics, 1993: 2(2): 153-174

[14] Kamiya T, Kusumoto S, Inoue K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code[J]. IEEE Trans. on Software Eng, 2002, 7(28): 654-670

[15] Basit H A, Jarzabek P S. Efficient token based clone detection with flexible tokenization[C] // ESEC/FSE'07. ACM Press, 2007: 513-516

[16] Wise M J. YAP3: improved detection of similarities in computer program and other texts[C] // Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education. 1996: 130-134

[17] L Jiang, Z Dong, E Chiu. Context-Based Detection of Clone-Related Bugs[M]. ACM Press, 2007: 55-64

[18] 王伟，苏小红，马培军，等. 标识符重命名不一致性缺陷的检测[J]. 哈尔滨工业大学学报，2011，43(1)：89-98

[19] Wang Tian-tian, Su Xiao-hong, Wang Yu-ying, et al. Semantic similarity-based grading of student programs[J]. Information and Software Technology, 2007, 49(2): 99-107

[20] 马培军，王甜甜，苏小红. 基于程序理解的编程题自动评分方法[J]. 计算机研究与发展，2009，46(7)：1136-1142

（上接第 518 页）

表 3　算法效率

编号	代码长度(Byte)	平均执行时间(ms)
1	197	13.45
2	1137	14.66
3	2127	18.95
4	2614	21.35
5	2960	21.55
6	3711	25.69
7	4088	27.21
8	5029	28.86
9	6017	29.81
10	6996	36.64
11	8395	40.31
12	8672	44.47
13	10504	49.61
14	11924	51.82
15	13588	58.19
16	14393	58.77
17	15265	63.48
18	17309	69.96
19	19256	77.15
20	20332	79.92

结束语　理论分析与实验验证表明，本算法可以在不影响代码分析的前提下，对 GCC 抽象语法树进行很大程度的化简，具有较低的时间复杂度，准确性和效率都得到了保障。经过化简后的 GCC 抽象语法树文件，可以更为方便地应用于代码分析领域，并在一定程度上提高相应算法的执行效率。

参 考 文 献

[1] 吴楠. 基于 GCC 的缓冲区溢出检测研究[D]. 哈尔滨：哈尔滨工程大学，2010

[2] 赵彦博. 基于抽象语法树的程序代码抄袭检测技术研究[D]. 呼和浩特：内蒙古师范大学，2010

[3] 张良德，赵彦博. 一种基于 GCC 抽象语法树的程序特征提取方法[J]. 电子技术与软件工程，2013(20)：269-270

[4] 江梦涛，荆琦. C 语言静态代码分析中的调用关系提取方法[J]. 计算机科学，2014，41(6A)：442-444

[5] Option Summary [EB/OL]. (2013-10-16) [2014-11-10]. http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Option-Summary.html

[6] 杨昌坤，许庆国. C 程序控制流程模型的提取技术与实现[J]. 计算机科学，2014，41(5)：208-214

[7] 李鑫，王甜甜，苏小红，等. 消除 GCC 抽象语法树文本中冗余信息的算法研究[J]. 计算机科学，2008，35(10)：170-172

[8] Deitel P, Deitel H. C 语言大学教程(第六版)[M]. 苏小红，等译. 北京：电子工业出版社，2012