第44卷 第7期
2012年7月

哈 尔 滨 工 业 大 学 学 报
JOURNAL OF HARBIN INSTITUTE OF TECHNOLOGY

Vol. 44 No. 7
July 2012

冗余代码缺陷检测方法

龚丹丹，王甜甜，苏小红，马培军

（哈尔滨工业大学 计算机科学与技术学院，150001 哈尔滨）

摘 要： 为解决冗余代码缺陷检测复杂度较高且检测精度较低的问题，设计并实现了基于控制结构的冗余代码检测模型. 通过对 TOKEN 序列建立复合语句结构信息表，精简了程序的控制依赖关系，并在此基础上对幂等操作、死代码以及冗余赋值3种冗余代码进行检测，有效降低了缺陷检测复杂度. 通过分析 Linux 开源代码表明，本模型可以快速的检测大规模程序，并且具有较低的误报率和漏报率. 因此本模型可以帮助程序员发现进而修正软件缺陷，维护软件可靠性.

关键词： 冗余代码；TOKEN 序列；代码标准化

中图分类号： TP311 **文献标志码：** A **文章编号：** 0367－6234(2012)07－0058－06

Redundancy detection based on control structure analysis

GONG Dan-dan, WANG Tian-tian, SU Xiao-hong, MA Pei-jun

(School of Computer Science and Technology, Harbin Institute of Technology, 150001 Harbin, China)

Abstract: To deal with the problems such as high complexity and low accuracy of redundancy detection, a model of redundancy detection based on control structure analysis is proposed and implemented. This paper predigests the complexity of control structure by establishing a compound node table for tokens, which reduces the complexity of redundancy detection, and then detects the idempotent operations, dead code and redundant assignment. Experimental results of the open source code of Linux show that this model can find redundant code accurately and also has a low time-complexity. With this model, it is very convenient for developers to detect and correct these kinds of defects, and thereby to further guarantee the software quality.

Key words: redundant code; TOKEN; code standardization

程序中的冗余代码意味着算法设计、代码实现中存在着问题. 冗余代码不仅影响软件的测试评估和运行效率，同时还存在着潜在的安全隐患，并由此而引发语义和逻辑缺陷，而这些缺陷很难被已有的软件缺陷检测工具检测到，因此，需要对程序中的冗余代码进行检测. 目前的冗余代码检测算法普遍误报率较高，很多学者针对缺陷误报率，进行了广泛的研究. 其中，符号执行类方法[1－5]成为当前热门的检测方法，但在遇到循环、过程调用、选择分支过多时，符号执行实现困难. 文献[6－9]根据区间运算理论提高检测精度，但目前现有的区间运算理论支持的数据类型单一，并不能解决非线性函数的区间运算和复杂条件表达式的区间运算. 同时，这些方法都是基于控制依赖图和数据依赖图，分析的复杂度高，并不适合测试大型软件. 本文具体分析了冗余代码不同类型所产生的误检原因，制定了相关的抑制误检的策略，有效提高了检测的精确度.

基于 TOKEN 序列的检测方法，由于其具有最佳的时空效率，已经被广泛用于重复代码检测. 基于 TOKEN 序列的重复代码检测是将程序表示成 TOKEN 序列，使用模式匹配技术检查代码间

收稿日期： 2011－06－21.

基金项目： 国家自然科学基金资助项目(61173021)；高等学校博士学科点专项科研基金资助项目(20112302120052, 20092302110040)；中央高校基本科研业务费专项资金资助项目(HIT. NSRIF. 201178).

作者简介： 龚丹丹(1982—)，女，博士研究生；
苏小红(1966—)，女，教授，博士生导师；
马培军(1963—)，男，教授，博士生导师.

通信作者： 龚丹丹，gongdandan0418@126.com.

的相似度，识别词法相似的程序代码，代表工具有 Duploc、NICAD、Dup、CCFinder、CP-Miner 等[10-14]. 这些方法的时间与空间开销较低，因此适合于分析大规模程序，然而它们只是在词法级别上对代码进行分析，并不能考虑语法或语义，仅仅只是处理程序的顺序结构，而无法处理选择和循环结构. 简单的结构代码改变就可使该方法失效，因此不适合直接在 TOKEN 序列的基础上检测冗余代码. 系统依赖图是程序的语义表示，能够充分表示程序的数据依赖、控制依赖信息以及函数调用信息，这适合于检测冗余代码缺陷，但因其系统依赖图的复杂度较高，并不适合于检测大规模程序.

针对上述分析，本文提出了精简的控制依赖关系，通过对 TOKEN 序列建立复合语句控制结构信息表，降低程序表示和分析的复杂度，以达到检测大型软件中的冗余代码的目的.

1　代码标准化

由于编程语言语法的多样性和代码实现的灵活性，相同的算法可能有多种不同的代码表达方式. 例如，有些程序元素可以用一条语句表示，也可以用多条语句构成的语句序列表示（int i,j; 等价于 int i; int j;），这种现象称为代码多样化. 识别这种语法表达形式多样但语义等价的代码，给程序分析带来了困难[15]，直接影响了缺陷检测的精确度. 程序标准化[16]是根据一系列标准化规则对程序进行语义等价的转换[17]的过程，目的是使语法表示不同但语义等价的程序有相同的系统表示，从而消除代码多样化，简化程序分析.

本文在 TOKEN 序列的基础上，根据一系列标准化规则进行结构语义等价的转换. 转换规则如下.

1）将表达式转换为只引用变量而不定义变量的形式，从而消除副作用. 例如：

for (i = j/k;;) {…} → i = j/k; for (i;;) {…}.

2）拆分变量声明语句. 例如：

int i, j; → int i; int j.

3）拆分变量声明与初始化语句. 例如：

int i = 0; →int i; i = 0.

4）拆分连续的赋值语句. 例如：

i = j = k; → j = k; i = j.

5）转换 + +、- -运算符. 例如：

n + +; → n = n + 1.

6）转换符合运算符（ + =、- =、* = / =、% =）. 例如：

n + = m; → n = n + m.

7）如果复合表达式中含有函数调用语句，则用临时变量替换该函数调用语句. 例如：

z = x + sqrt(y); → t = sqrt(y); z = x + t.

8）如果函数调用语句中的实参是复合表达式，则用临时变量替换该复合表达式. 例如：

i = fun(j + k) → t = j + k; i = fun(t).

9）如果 return 语句返回的是复合表达式，则用临时变量替换该复合表达式. 例如：

return i + j; → t = i + j; return t.

在消除代码多样化后大多数语法不同但语义等价的程序具有相同的 TOKEN 表示. 这为冗余代码缺陷检测奠定了良好的基础，不仅可以简化程序分析，而且可以消除由于代码多样化所产生的误检.

2　复合语句控制结构信息表

复合语句控制结构信息表（CNT）记录了各复合语句的开始 - 结束位置等重要信息，按照 CNT 遍历程序的 TOKEN 序列，可以准确控制程序的扫描路径，CNT 只记录复合语句信息，不需要分析整个程序的控制依赖关系，其复杂度远远低于控制依赖图.

冗余代码缺陷检测，需要考虑程序的控制依赖关系，因此，不能直接利用传统的 TOKEN 序列进行检测.

针对上述问题，本文针对每个函数建立复合语句控制结构信息表，存储复合语句的结构信息. 根据结构信息表扫描 TOKEN 序列，充分考虑了程序的控制信息，节省了时间与空间，以达到应用于大规模程序的目的.

定义 1（复合语句控制结构信息表（Compound node table，CNT））　复合语句控制结构信息表是记录程序 P 中的所有复合语句（选择、循环）的开始、结束位置等关键信息的集合，它能够精简的记录程序的所有控制依赖关系，其结构定义如图 1 所示.

```
typedef struct          //记录复合语句起止位置
{
  int id;               //复合语句id号
  CString name;         //复合语句名
  int key;              //关键字
  CString file;         //所属文件
  int beginline;        //复合语句开始行
  int endline;          //复合语句结束行
  int beginID;          //复合语句开始token位置
  int endID;            //复合语句结束token位置
  int ifendID;          //复合语句内部结束token位置
  int ifendline;        //复合语句内部结束行
} COMPNODE;
```

图 1　复合语句控制结构信息表结构

对于多分支的选择结构,不仅应该记录选择结构的结束位置(endID)和结束行(endline),还应该记录各个分支内部的结束位置(ifendID)和结束行(ifendline),以便进行精确的路径分析. 例如图 2 所示,多分支选择结构 if(第 13 行),其内部分支结束位置 ifendline = 16,总分支结束位置 endline = 20;对于第 33 行的 else 分支,其内部分支结束位置即为总分支的结束位置,即 endline = ifendline = 20. 循环结构无需考虑多分支情况,因此 for 的 endline 与 ifendline 的值相同.

```
D:\aa.c
11 for (n=1; n>1; n++)
12 {
13     if( m<6)
14     {
15         sum= +x; y
16     }
17     else
18     {
19         f=aver(a, b);
20     }
21 }
```

id	1	2	3
name	for	if	else
key	CN_FOR	CN_IF	CN_ELSE
file	D:\aa.c	D:\aa.c	D:\aa.c
beginline	11	13	17
endline	11	20	20
beginID	…	…	…
endID	…	…	…
ifendID	…	…	…
ifendline	11	16	10

图 2 复合语句控制结构信息表实例

3 冗余代码缺陷检测

3.1 方法框架描述

冗余代码检测模型如图 3 所示,主要包括 3 个部分:

1)将程序转换为 TOKEN 序列,并在其基础上进行代码标准化,简化程序分析的同时消除由于代码多样化所产生的误检,进而在标准化的 TOKEN 序列基础上检测幂等缺陷.

2)TOKEN 序列缺少程序的结构化信息,因而不能很好的表示变量的依赖关系. 系统依赖图的复杂度较高,不适合分析大规模程序. 针对此点,本文建立 CNT,精简了控制依赖关系,即能保证时间与空间复杂度,又能充分表示程序的控制依赖关系.

3)根据 CNT,在标准化后的 TOKEN 序列上检测死代码和冗余赋值缺陷.

最后根据缺陷检测结果,输出缺陷检测报告.

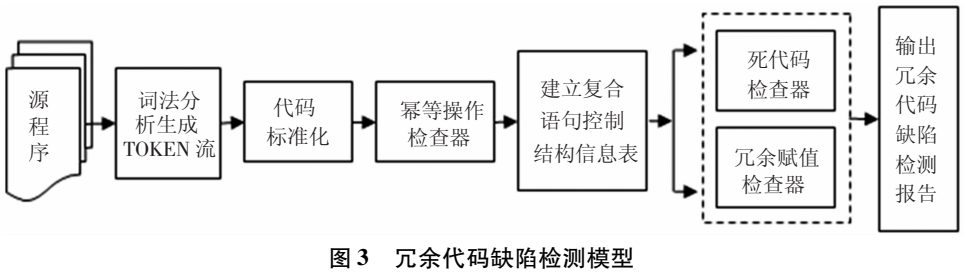

图 3 冗余代码缺陷检测模型

3.2 幂等操作检查器

幂等操作包括下列 3 种情况:

1) 赋值给自己($x = x$);

2) 被自身除(x/x);

3) 自己与自己的位操作($x \mid x$) 或($x\&x$).

本检查器直接扫描程序的 TOKEN 序列,当所扫描的 TOKEN 为运算符“ = ”、“/”、“|”、“&”任一运算符时,比较运算符左右两端的 TOKEN 是否相等,如果相等,报告此缺陷. 图 4 为利用幂等操作检查器所检查到的缺陷.

```
1 /*linux2.6.6/arch/ppc64/kernel/sys_ppc32.c
2  err |= __get_user(karg->ca_export.ex_anon_uid,
3       &arg32->ca32_export.ex32_anon_uid);
4  err |= __get_user(karg->ca_export.ex_anon_gid,
5       &arg32->ca32_export.ex32_anon_gid);
6      /**************** bug ****************/
7  karg->ca_export.ex_anon_uid = karg->ca_export.ex_anon_uid;
8  karg->ca_export.ex_anon_gid = karg->ca_export.ex_anon_gid;
```

图 4 利用幂等操作检查器可以检测出的软件缺陷

3.3 死代码检测器

所谓检查器,是指检测在程序运行时不可到达的代码. 通过在 Linux 中的实验表明,此类缺陷的产生原因主要是由于 break 和 return 所导致的奇怪控制流. 例如:break 后面但和 break 同属一个块的执行语句为死代码;所有条件分支都包含 return,则后面的代码将不会被执行等等.

该检查器涉及程序的控制信息,因此,根据 CNT,在 TOKEN 序列的基础上进行控制结构分析. 对每条路径,标记所有可由路径所到达的语句. 对于未标记的语句给出错误信息. 为了提高效率,本检查器以函数为单位进行检测,并跳过宏定义,图 5 为死代码检测具体算法.

此算法的关键之处在于按复合语句结构列表计算结束位置,充分考虑选择、循环的嵌套情况,根据不同的情况计算 TOKEN 序列的扫描跳转位置,即根据不同情况选择 ifendID 和 endID. 此检查器找到很多明显的错误,并且无误检.

图 6、7 为利用死代码检查器所检查到的缺陷. 图 6 中第 6 行 return 语句后的语句不会执行,图 7 中循环中包含的 if-else 语句中两个分支均跳出循环,则后面的第 13 ~ 18 行语句不会被执行.

```
算法：死代码检查器算法
输入：源程序 TOKEN 序列                    输出：死代码
声明：linenum[ln]记录可到达的行号;
begin
    建立函数复合语句控制表
  while(TOKEN 序列中还有未被读取的元素)
    if(linenum[ln]未记录该 TOKEN 元素行号)
          将元素行号保存到 linenum[ln]中;
    endif
    if(读取元素为宏定义)
      while(宏定义结束)
        if(linenum[ln]未记录该 TOKEN 元素行号)
          将元素行号保存到 linenum[ln]中;
        endif
      endwhile
    endif
    if(读取元素为"return"或"break")
      查找复合语句结构列表
      if(读取元素不在复合语句结构列表中)
        跳到 TOKEN 序列结束位置;
      endif
      if(读取元素在复合语句结构列表中)
        按复合语句结构列表计算结束位置
        跳转到所计算的结束位置
      endif
    endif
  endwhile
  while(TOKEN 序列中还有未被读取的元素)
    if(该 TOKEN 元素行号不在 linenum[ln]中)
      输出 TOKEN 元素;
    endif
  endwhile
end
```

图 5　死代码检查器算法

```
1 /*linux2.6.6/arch/mips/kernel/sysirix.c
2 case SGI_GETNVRAM:
3   {   char *name = (char *) regs->regs[base+5];
4       char *buf = (char *) regs->regs[base+6];
5       char *value;
6       return -EINVAL; /* til I fix it */
7        /*bug : cannot be reached*/
8       retval = verify_area(VERIFY_WRITE, buf, 128);
9       if (retval)
10          break;
11      value = prom_getenv(name); /* PROM lock? */
12      if (!value) {
13          retval = -EINVAL;
14          break;
15      }
16      copy_to_user(buf, value, 128);
17      retval = 0;
18      break;
19  }
```

图 6　死代码检测实例 1

```
1 /*linux2.6.6/arch/ mips/kernel /irixelf.c
2  for (c = (struct NCR53c7x0_cmd *) hostdata->running_list; c; c = (struct
NCR53c7x0_cmd *) c->next)
3  {
4    if (c->cmd->SCp.buffer) {
5      printk ("scsi%d : loop detected in running list!\n", host->host_no);
6      break;
7      }
8    else
9     {
10    printk ("Duh? Bad things happening in the NCR driver\n");
11    break;
12    }
13   /* bug: cannot be reached */
14   c->cmd->SCp.buffer = (struct scatterlist *) list;
15  list = c->cmd;
16  if (free)
17    {  c->next = hostdata->free;
18     hostdata->free = c;
19    }
20  }
```

图 7　死代码检测实例 2

3.4　冗余的赋值语句检查器

该检查器跟踪变量的生存周期，在每个赋值语句处向前跟踪所有路径上的该变量. 如果变量在退出所在域或被重新赋值前没被使用，并且变量并非作为函数参数返回值，则发送错误信息.

该检查器涉及程序的控制依赖分析，则仅在 TOKEN 序列的基础上扫描必然带来大量误检. 因此，本文根据 CNT，在 TOKEN 序列的基础上进行控制结构分析，在最佳的时空效率下保证了检测精度.

此检查器的误检大多来自于保守编程. 本文分析了保守编程的具体情况，根据不同的情况，制定不同的策略：

1）特殊赋值或前面有类型定义：p = head；int i = 5；此类情况为保守编程，不给出错误信息.

2）第 1 次赋值（除情况 1 外）. 可能是保守编程为变量赋初值，也可能是直接为变量赋值，对于此情况给出警告信息. 具体算法如图 8 所示.

```
算法：冗余的赋值语句检查器算法
输入：源程序 TOKEN 序列                    输出：冗余的赋值语句行号
begin
    建立函数复合语句控制表
  while(TOKEN 序列中还有未被读取的元素)
    if(找到"x=")
      if("x="是第 1 次赋值)
          给出警告信息;
      endif
      if("x="后面使用自身" ||"x="后面赋值为特殊值，即保守编程||"x="前面有类型定义)
          退出，不记录行号
      endif
      查找复合语句结构列表
      if("x="在 for、while、 do…while、if 的条件中)
          退出，不记录行号
      endif
      if("x="在 for、while、 do…while、if、switch 模块中)
        在模块内部查找是否有"x="
        if(模块内部找到"x=")
            记录行号，退出
        else if(模块内部找到使用变量"x")
            退出，扫描下一 TOKEN 元素
        else if("x"自身在循环条件中或"x="后面使用了循环条件中的变量)
            退出，扫描下一 TOKEN 元素
        else
            根据按复合语句结构列表计算结束位置并跳转
       endif
       while(未扫描到 TOKEN 序列结束位置)
        if(找到"x=")
          记录行号，退出
        else if(找到使用变量"x")
          退出，扫描下一 TOKEN 元素
       endwhile
       if("x"是全局变量||"x"函数参数)
        退出，扫描下一 TOKEN 元素;
       endif
    endif
  endwhile
end
```

图 8　冗余的赋值语句检查器算法

对于赋值语句位于循环结构中的情况，不仅要考虑"x =" 中的"x" 以及后面赋值的变量直接使用了循环变量，还要检测"x" 以及后面赋值的变量间接使用循环变量的情况，以减少误检.

图 9 中列出了此检查器检测的 Linux 代码冗余缺陷，第 5 行第 1 次为变量赋值，给出警告信息，第 7 行变量赋值后未使用，给出错误提示.

```
1 /*linux2.6.6/arch/arm/kernel/semaphore.c
2 void __sched __down(struct semaphore * sem)
3 {struct task_struct *tsk = current;
4  DECLARE_WAITQUEUE(wait, tsk);
5  tsk->state = TASK_UNINTERRUPTIBLE;  // 第一次赋值未使用，给出警告
6  --
7  tsk->state = TASK_UNINTERRUPTIBLE;  // bug
8  spin_lock_irq(&semaphore_lock);
9 }
```

图 9　冗余的赋值语句检测实例

4　实验结果与分析

4.1　实际开源代码缺陷检测结果分析

本文对开源代码 Linux-2.6.6 中的部分模块进行了检测. 本文实验条件为:Intel 酷睿 2 双核 T5670,1.80 GHzCPU,1GB 内存,开发用工具为 VC + +6.0. 表 1 ~ 表 3 列出了实际开源代码缺陷检测的实验结果.

表 1　幂等缺陷检测结果统计

子模块名称	C 文件总数	源代码总行数	检测时间/s	疑似缺陷数	人工确认的缺陷数	误检率/%
Linux-2.6.6/arch/arm	240	49 480	8.735	12	12	0
Linux-2.6.6/arch/ia64	129	66 837	25.952	9	9	0
Linux-2.6.6/arch/mips	453	91 355	15.563	22	22	0
Linux-2.6.6/arch/ppc、ppc64 和 cris	412	159 718	42.422	24	24	0

表 2　死代码缺陷检测结果统计

子模块名称	C 文件总数	源代码总行数	检测时间/s	疑似缺陷数	人工确认的缺陷数	误检率/%
Linux-2.6.6/arch/arm	240	49 480	11.969	2	2	0
Linux-2.6.6/arch/ia64	129	66 837	47.844	2	2	0
Linux-2.6.6/arch/mips	453	91 355	21.937	3	3	0
Linux-2.6.6/arch/ppc、ppc64 和 cris	412	159 718	63.594	10	10	0

表 3　冗余赋值缺陷检测结果统计

子模块名称	C 文件总数	源代码总行数	检测时间/s	警告	人工确认	疑似缺陷数	人工确认的缺陷数	误检率/%
Linux-2.6.6/arch/arm	240	49 480	12.578	37	37	21	21	0
Linux-2.6.6/arch/ppc、ppc64 和 cris	412	159 718	227.355	58	58	39	39	0

4.2　人工注入缺陷检测结果分析

为了统计漏检率,本文对部分 Linux 源代码进行了人工注入缺陷实验. 表 4 ~ 表 6 列出了人工注入缺陷检测实验结果.

通过在 Linux 源代码中实验表明,本文所提出的冗余代码缺陷检测算法可以快速的识别大规模程序中的幂等操作、死代码以及冗余赋值缺陷,并且检测精度高,误报率及漏报率均为 0. 幂等缺陷检查器直接在 TOKEN 序列基础上进行扫描,时间复杂度最低. 而死代码和冗余赋值缺陷检测以函数为单位,在 TOKEN 序列的基础上,利用 CNT 查找程序的控制依赖关系,时间复杂度略有提高. 冗余赋值检测器为了有效抑制误检,对于变量定义后第 1 次赋值,有可能是保守编程,给出警告. 若变量不是第 1 次赋值,报告冗余赋值缺陷.

表 4　人工注入幂等缺陷检测结果统计

输入代码规模	人工注入缺陷数量			显式恒等操作检查器检测出的人工注入的缺陷数量			漏检率/%		
	自赋值	被自身除	自身的位操作	自赋值	被自身除	自身的位操作	自赋值	被自身除	自身的位操作
$<10^4$ 行	20	30	20	20	30	20	0	0	0
$>10^5$ 行	20	30	20	20	30	20	0	0	0

表 5　人工注入死代码缺陷检测结果统计

输入代码规模	人工注入缺陷数量	死代码检查器检测出的人工注入的缺陷数量	漏检率/%
$<10^4$ 行	20	20	0
$>10^5$ 行	30	30	0

表 6　人工注入冗余赋值缺陷检测结果统计

输入代码规模	人工注入缺陷数量	冗余赋值检查器检测出的人工注入的缺陷数量	漏检率/%
$<10^4$ 行	20	20	0
$>10^5$ 行	30	30	0

5　结　论

1）提出了代码标准化规则，且消除代码多样化，并简化程序分析，有利于消除缺陷检测中因代码多样化产生的误检.

2）本文提出了 CNT，精简了程序中控制依赖关系的表示.

3）设计并实现了基于控制结构的冗余代码缺陷检测模型，对程序中的幂等操作、死代码以及冗余赋值 3 种冗余代码进行检测. 该模型针对各种可能产生误检的原因进行了分析，制定了相关的抑制误检的策略. 通过实验证明，本模型可以快速检测大规模程序，并且具有较低的误报率和漏报率.

参考文献：

[1] ZHANG Jian. Constraint solving and symbolic execution [C]//Proceedings of the Verified Software: Theories, Tools, Experiments. Heidelberg, Berlin: Springer-Vierlag, 2005: 539 – 544.

[2] XU Xiao, ZHANG Xiaosong S, LI Xiongda. New approach to path explosion problem of symbolic execution [C]//Proceedings of the 2010 First International Conference on Pervasive Computing, Signal Processing and Applications. Washington, DC: IEEE Computer Society, 2010: 301 – 304.

[3] COHEN M B, DWYER M B, SHI Jiangfan. Exploiting constraint solving history to construct interaction test suites[C]//Proceedings of the Testing: Academia and Industry Conference-Practice And Research Techniques. Washington, DC: IEEE Computer Society, 2007:121 – 130.

[4] PAPADAKIS M, MALEVRIS N. Automatic mutation test case generation via dynamic symbolic execution [C]//Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering. Washington, DC: IEEE Computer Society, 2010: 121 – 130.

[5] PAPADAKIS M, MALEVRIS N. A symbolic execution tool based on the elimination of infeasible paths[C]// Proceedings of the 2010 Fifth International Conference on Software Engineering Advances. Washington, DC: IEEE Computer Society, 2010: 435 – 440.

[6] 王雅文，宫云战，肖庆，等. 区间运算在软件缺陷检测中的应用[C]. 苏州：第五届中国测试学术会议论文集，2008：51 – 52.

[7] 王雅文，宫云战，肖庆，等. 扩展区间运算的变量值范围分析技术[J]. 北京邮电大学学报，2009，32(3)：36 – 41.

[8] 王志言，刘椿年. 区间算术在软件测试中的应用[J]. 软件学报，1998，9(6)：438 – 443.

[9] GHODRAT M A, GIVARGIS T, NICOLAN A. Expression equivalence checking using interval analysis[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2006, 14(8):830 – 842.

[10] DUSCASSE 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: IEEE Computer Society, 1999:109 – 118.

[11] ROY C K, CORDY J R. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization[C]//Proceedings of the 2008 the 16th IEEE International Conference on Program Comprehension. Washington, DC: IEEE Computer Society, 2008:172 – 181.

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

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

[14] LI Zhenmin, LU Shan, 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.

[15] RIEGER M. Effective clone detection without language barriers[D]. Switzerland: Dissertation, University of Bern, 2005.

[16] De JONGE M, VISSER E, VISSER J. XT: a bundle of program transformation tools system description[J]. Electronic Notes in Theoretical Computer Science, 2001, 44(2): 79 – 86.

[17] VISSER E. A survey of strategies in rule-based program transformation systems[J]. Journal of Symbolic Computation, 2005,40(1):831 – 873.

（编辑　张　红）