CN43-1258/TP
ISSN 1007-130X

计算机工程与科学
COMPUTER ENGINEERING & SCIENCE

2012 年第 34 卷第 2 期
Vol. 34, No. 2, 2012

文章编号:1007-130X(2012)02-0067-06

基于 MYGCC 的编程规则检查算法研究*
Research on a MYGCC-Based Algorithm for Checking Programming Rules

李　锋,文艳军,齐治昌,陆赛因
Li Feng, WEN Yan-jun, QI Zhi-chang, LU Sai-yin
(国防科学技术大学计算机学院,湖南 长沙 410073)
(School of Computer Science, National University of Defense Technology, Changsha 410073, China)

摘　要:MYGCC 是一个编程规则检查工具,其目前的检查算法存在局限性,不能完整地展示违反编程规则的程序路径。本文提出并实现了一种改进的编程规则检查算法,可以弥补上述的局限性。实验表明改进算法是有效的,此改进有助于用户更准确地定位错误位置,方便对编程错误的修正。

Abstract: MYGCC is a tool for checking programming rules, and its current checking algorithm has defects. That is, it can not show the whole control flow paths that violate the checking rules. To eliminate the defects, this paper presents an improved algorithm to the original one of MYGCC. Experiment shows that the algorithm is effective. This improvement can help users to find the position of bugs more accurately, and benefit bug fixing.

关键词:编程规则检查;静态分析;可扩展编译器

Key words: programming rule checking; static analysis; extensible compiler

doi: 10.3969/j.issn.1007-130X.2012.02.013

中图分类号:TP311　　**文献标识码**:A

1　引言

程序的静态分析工具分两类,一类是对描述程序的语法和语义进行检查,另一类是对用户自定义的编程规则进行检查。前者是编译过程顺便完成的,后者需要专门的软件工具。MYGCC 是一个程序静态分析工具,用于用户自定义软件源代码编程规则的检查。它是在 GNU C Compiler (GCC)编译器的基础上扩展而成的,使得源程序静态分析与编译过程紧密结合,在编译的同时对源程序执行用户自定义的编程规则检查[1,2]。MYGCC 有助于在软件编程实现阶段发现更多的潜藏错误,从而尽早纠正错误,减少开发维护成本。这类静态分析技术正被广泛地用于软件开发实践,是一种提高软件质量的有效手段。

MYGCC 算法存在局限性,对于源程序错误只能定位到错误路径的终点的行号。本文提出并实现了一种改进的编程规则检查算法,可以弥补上述 MYGCC 的局限性。按照这个改进算法,并在 MYGCC 中做了实现并形成新版本 MYGCC-B,测试表明此算法是有效的。

本文第 2 节描述了现有算法的局限性;第 3 节论述了改进算法,并分析了其正确性;第 4 节描述了使用改进后的 MYGCC-B 进行实验分析;最后是对本文的小结。

2　MYGCC 的局限性

以一个判断闰年的 C 源程序为例,对上述

* 收稿日期:2010-01-29;修订日期:2010-04-15
基金项目:国家自然科学基金重点项目(60803042,90818024,60970035)
通讯地址:410073 湖南省长沙市国防科学技术大学计算机学院
Address: School of Computer Science, National University of Defense Technology, Changsha, Hunan 410073, P. R. China

MYGCC 的局限性进行描述。源程序文件 test.c 中描述的是一个判断某年是否为闰年的函数程序，如果是闰年则返回 1，否则返回 0。为了描述上述 MYGCC 的局限性，在程序中特意插入了内存分配操作。C 源程序文件 test.c 如下：

```
1 #include <stdlib.h>
2 int isleapyear(int year)
3  {
4  int *p;
5  p = (int *)malloc(sizeof(int));/*后面对指针P的使用，存在对动态内存的不安全使用*/
6  if(year%400 == 0)
7    {
8    printf("%d can be divided exactly by 400\n",year);
9     *p = 1;
10  }
11 else
12  {
13   if(year%4 == 0 && year%100 != 0)
14     *p = 1;
15   else
16     {
17       if (p != NULL)
18          *p = 0;
19     }
20   }
21   return *p;
22  }
```

规则文件 test.chk 中描述了一条自定义的编程规则，此规则的作用是检查程序是否存在对动态分配内存的不安全使用[3]。它是用专用语言编写的，其含义是：对于用 malloc 函数分配的内存，在使用之前必须先判断分配动作是否真正成功。规则文件 test.chk 如下：

```
condate malloc_deref {
  from "%X = malloc (%_)" # any malloc
  to "%_ = %X->%_" or "%X->%_ = %_"
  or "*%X = %_" or "%_ = *%X"
  avoid "%X = %_" or +"%X != 0B" or -"%X == 0B"
} warning("Unsafe dereference of X after malloc");
```

MYGCC 首先对 Condate 文件进行词法分析和语法分析，提取 check 文件中的规则后交给 Condate Checker。Condate Checker 以这些规则对由前端接口生成的 GIMPLE 进行检查，如果发现有与规则相匹配的情形发生，则输出具体的出错信息。检查完成后，将 GIMPLE 交予后端接口继续进行处理[1,4]。

Condate 规则语言是以 BNF 语法定义的语言。基本语法如下[5,6]：

S→from D [to D [avoid D]] (1)

D→E | E or E (2)

E→P|+P|-P (3)

P→"(%V | lit)_" [|expr] (4)

在这里 S 是开始符，D 代表一个析取范式，E 是一个边范式，P 是一个基本范式，V 是一个范式变量，lit 是文字代码片断，expr 是一个编译器可执行表达式。在规则 malloc_deref 中，%X 可以与一个指针变量匹配，%_可以与常量或者变量匹配。

其中，condate 关键字指出了一条规则的开始，紧接着的 malloc deref 是此条规则的名字[2,5,6]。

“from”关键字后描述该条规则所描述路径的开头语句。MYGCC 在检查过程中将会对源代码中各条语句与“from”所描述的语句进行匹配，匹配成功则说明该条语句为可能存在边的开头。“to”关键字后描述的是该条规则所描述路径的结束语句。MYGCC 的检查过程中如出现与“to”匹配的语句，则说明在控制流图上自“from”至“to”找到了一条待检查边。

“avoid”关键字后描述在路径搜索中应该避开的语句或分支。这些语句或分支应该在“from”和“to”之间出现而实际上跳过的语句。“+”和“-”是对“avoid”项的补充说明，用以说明当源代码中出现 if 选择结构时是避开 then 分支还是避开 else 分支。

“warning”关键字后的字符串记录该条规则所代表错误的描述，当检查中出现与规则匹配的情况，MYGCC 将告警。

用 MYGCC 对 C 源程序 test.c 进行检查，虽然 MYGCC 在代码中查出了错误，但它只能显示错误路径的终点语句行号，不能显示整条路径，定位不够精确，不便于错误的修正。其中符号“<-”表示替换的意思，如“X<-p”表示指针 p 替换 X。MYGCC 对 C 源程序 test.c 进行检查结果如下：

```
$ mygcc /root/test/test.c --tree-checks=/root/test/test.chk --tree-checks-verbose
/root/test/test.c: In function 'isleapyear':
/root/test/test.c:14: warning: user-defined warning malloc_deref: Unsafe dereference of X after malloc.
/root/test/test.c:14: instance = {X<-p},
/root/test/test.c:14: reached: *p = 1.
/root/test/test.c:21: warning: user-defined warning malloc_deref: Unsafe dereference of X after malloc.
```

/root/test/test.c:21: instance = {X←p},

/root/test/test.c:21: reached: D.2143 = *p.

/root/test/test.c:9: warning: user-defined warning malloc_deref: Unsafe dereference of X after malloc.

/root/test/test.c:9: instance = {X←p},

/root/test/test.c:9: reached: *p = 1.

3　对 MYGCC 编译器的改进

3.1　改进算法的描述

MYGCC 的原检查算法的核心思想是从满足“from”条件的语句出发,使用深度优先的方法遍历程序控制流图,且遍历过程中避开“avoid”指明的条件的分支,在此过程中如果发现一个满足“to”条件的语句可达,那么就说明发现了一条违规路径。在此过程中,只使用一个“待扩展节点栈”来记录所有需要进一步扩展的节点,所有已经扩展过的节点(即所有有价值的后继都已入栈的节点)是不在栈内留存的,都被弹了出来。这些从栈中弹出的已检查节点记录了从“from”开始到“to”的详细路径,因此该栈中缺乏许多历史信息。

改进思路是通过增加一个“历史节点栈”来收集那些在检查过程中从原待扩展节点栈中弹出的节点,并且随着遍历的前进或后退来同步增减此历史节点栈,从而有效记录关键的历史信息。这样当发现违规情况时,只需将历史节点栈输出来即可重现从“from”到“to”的路径上每一条语句的细节。整个过程如图 1 中的控制流图所示。其中深色背景的部分是新增的。

在图 1 中,Stack 代表原来的栈,存储所有带扩展的节点。为了保存自 Stack 中弹出的历史节点来重现自“from”到“to”的路径,需另建立一个堆栈 HisStack,其中节点的类型为一个结构体,即 struct CompoundNode { cfg_node opnode; int mark;}。图中的 pHisStack 为堆栈 HisStack 的栈顶指针,pHisStack→opnode 用于保存自 Stack 弹出的节点,pHisStack→mark 用来标示 HisStack 栈顶元素在 Stack 有几个后继节点。若 pHisStack→mark 值为 1,则表示 Stack 的栈顶为 HisStack 栈顶的后继节点;若 pHisStack→mark 值为 2,则表示 Stack 的栈中前两个元素都为 HisStackz 栈顶的后继节点,其它依此类推。

图 1 表明,MYGCC-B 采用深度优先的搜索方式,先是从 Stack 中弹出一个节点 node,并从中得到它的后续节点 succ_node,根据 succ_node 的情况决定是否将 succ_node 压入 Stack。改进算法后,对 succ_node 进行检查的同时,将对 HisStack

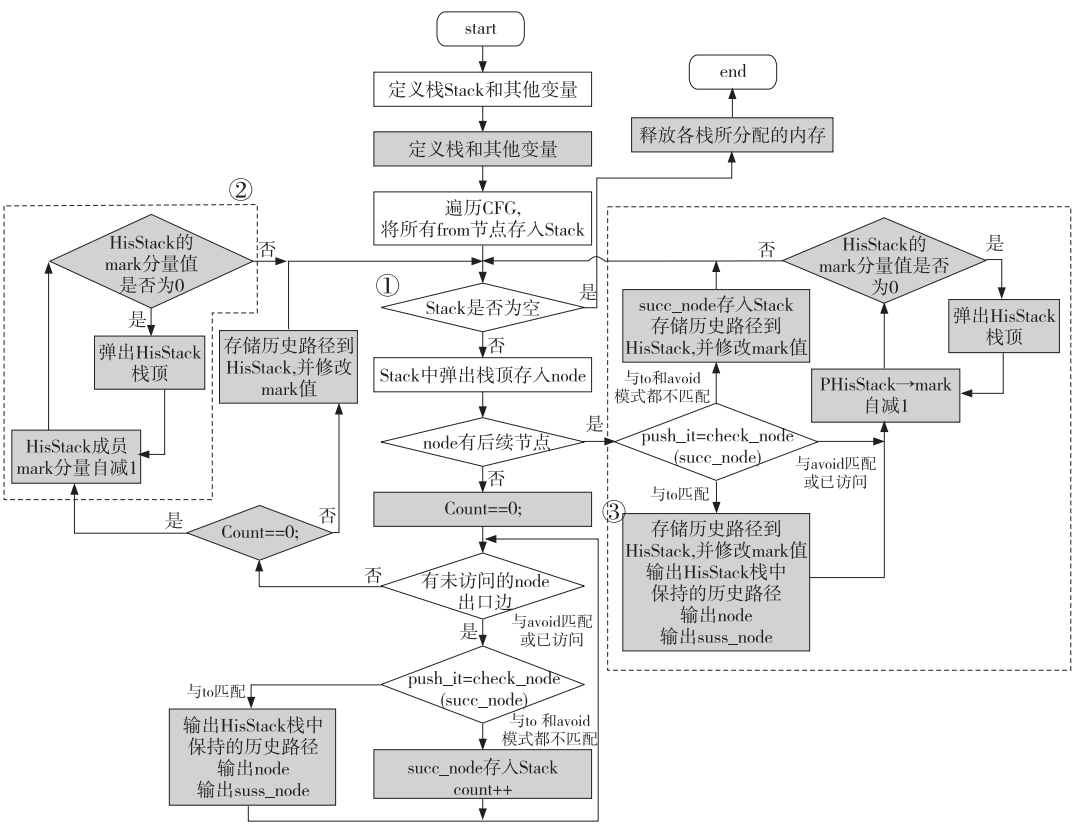


图 1　改进后的算法流程图

进行下列操作：

①若 succ_node 为 node 的直接后继节点，它未被访问，与 to 和 avoid 都不匹配，则需将 succ_node 压入 Stack，以便再继续搜索 succ_node 的后续节点，同时需将 node 压入 pHisStack→opnode，pHisStack→mark 值加 1，以记录搜索路径。若 succ_node 为 node 的出口边后继节点，同样将 succ_node 压入 Stack，同时 count 值加 1，用于计算 node 有几条出口边的后继节点被压入 Stack。

②若 succ_node 未被访问，但与 to 匹配，说明在控制流图中找到了一条自 from 开始，而不经过 avoid，就到达 to 的路径(from_node→…→node→succ_node)。若 succ_node 为 node 的直接后继，则需将 node 压入 pHisStack→opnode，同时 pHisStack→mark 值加 1，此时 HisStack 中存储了自 from_node 至 node 的所有节点。逆序输出 HisStack 中的所元素的 opnode 成员内容后再输出 succ_node，即可重现自“from”到“to”的路径。之后，HisStack 进行弹栈到 pHisStack→mark 值自减 1 后不为 0，停止弹栈。若 succ_node 为 node 的出口边后继节点，则逆序输出 HisStack 中的所元素的 opnode 成员内容后再输出 node、succ_node 值，即可重现自“from”到“to”的路径，之后继续判断其它边情况。

③若 succ_node 未被访问，但与 avoid 匹配，则不会将 succ_node 压入 Stack。若 succ_node 为 node 的直接后继，则 HisStack 需进行弹栈，直到 pHisStack→mark 值自减 1 后不为 0，停止弹栈。若 succ_node 为 node 的出口边后继节点，则继续判断其它边情况。

④若 succ_node 已被访问，和情况③一样处理。

对于 succ_node 为 node 的出口边后继节点情况，在 MYGCC-B 检查完所有的 node 出口边后继节点后，需要判断 count 是否为 0，不为 0，则说明有 node 边的后继节点被压入 Stack，此时需要把节点(node,count)压入 HisStack 栈。若 count 值为 0 则需要对 HisStack 进行弹栈，HisStack 每次弹栈前都需要先把 pHisStack→mark 自减 1，若 pHisStack→mark 自减后为 0，则把此时的 HisStack 栈顶弹出废弃，并将 HisStack 的新栈顶元素的 mark 成员再减 1，若也为 0 则将该新栈顶元素也弹出，如此反复，直到 mark 成员减 1 后不为 0 则停止弹栈。

3.2 改进算法的实例分析

下面分析使用改进后的 MYGCC-B 对 test.c 进行检查时，在几个关键节点时对应的堆栈 Stack、HisStack 状态变化。

(1)MYGCC-B 以深度搜索的方式遍历控制流图，检查完节点“if(year%400 ==0)”，进入图 1 流程图的程序块①前堆栈 Stack、HisStack 值如图 2 所示，图中 HisStack 的栈顶 mark 分量值为 2 表示此时栈顶 opnode 分量在栈 Stack 中有两个后继节点“if(year%4 == 0 && year%100 ! = 0)”和“printf(“%d can be divided exactly by 400\n”, year)”；而 HisStack 中栈底 mark 分量值为 1 表示栈底的 opnode 分量“p = (int *)malloc(sizeof(int))”在 HisStack 中有一个后继节点“if(year%400 ==0)”。从 Stack 中弹出“if(year%4 == 0 && year%100 ! = 0)”节点，检查它的 then 分支时，后继节点语句“*p = 1”与 to 匹配，找到第一条从 from 到 to 而未经过 avoid 的路径。MYGCC-B 检查完“if(year%4 == 0 && year%100 ! = 0)”的后继节点后，在进入图 1 中流程图的程序块①前堆栈 Stack、HisStack 值如图 3 所示，可见 HisStack 的栈顶元素的 mark 分量为 1，表示对应的 opnode 分量“if(year%4 == 0 && year%100 ! = 0)”在 Stack 栈中有一个后继节点，即“if(p! = NULL)”。而 HisStack 栈中间元素的 mark 分量为 2，表示对应的“if(year%400 ==0)”节点在这两个栈中有两个直接后继，即 Stack 栈中的节点“printf(“%d can be divided exactly by 400\n”, year)”和 HisStack 中的“if(year%4 == 0 && year%100 ! = 0)”。可见 MYGCC-B 保持了 Stack 栈顶为 HisStack 的栈顶 opnode 分量的后继节点这一性质。

if(year%4 == 0 && year%100 != 0)
Printf(“%d can ... year);

a :堆栈 Stack

if(year%400 == 0)	2
p = (int *)malloc(sizeof(int));	1
opnode	mark

b: 堆栈HisStack

图 2 Stack、HisStack 值 1

(2)检查“if (p ! = NULL)”的 else 边后继节点语句“return *p”与 to 匹配，找到第二条从 from 到 to 而未经过 avoid 的路径。而此时 node 节点的 then 边因为“p ! = NULL”与 avoid 项的“+"%X ! = 0B""匹配不对 then 边搜索，因为此项已经对内存分配是否成功作了判断，所以不存在对分配的内存不安全使用。MYGCC-B 对此 node 节点的检查完成，此时需进入图 1 中流程图的程序

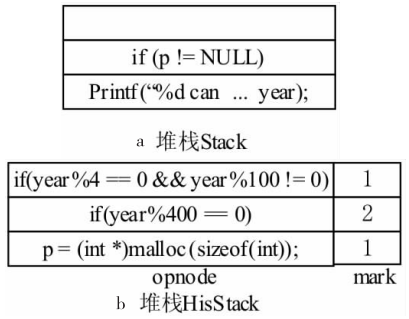


图 3　Stack、HisStack 值 2

块②对堆栈 HisStack 进行弹栈操作,出了程序块②,进入程序块①前堆栈 Stack、HisStack 值如图 4 所示。可以看到保持了 Stack 的栈顶为 pHisStack→opnode 的后继节点这一性质,此时的 HisStack 栈顶 mark 分量值为 1,说明此时 HisStack 栈顶 opnode 分量在 Stack 中只有一个后继节点。

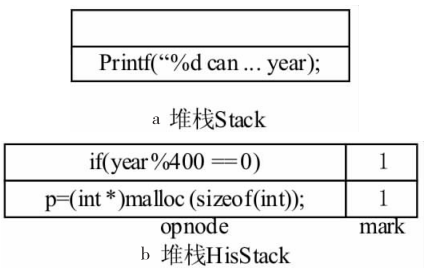


图 4　Stack、HisStack 值 3

(3)检查到 node 节点为语句"printf("%d can be divided exactly by 400\n", year)",因为 node 的后继节点为同一基本块内的后继节点,所以 MYGCC-B 需进入图 1 中流程图的程序块③进行处理。检查发现 node 的后继节点(* p = 1)与 to 匹配,说明此时找到第三条从 from 到 to 而未经过 avoid 的路径。此时,MYGCC-B 对控制流图遍历完成,出了程序块③后堆栈 Stack、HisStack 值全部清空。堆栈 Stack、HisStack 值如图 5 所示。

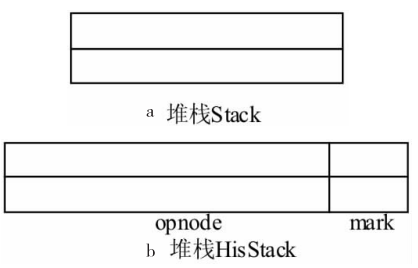


图 5　找到第一个与 to 匹配时 Stack、HisStack 值

4　改进后的 MYGCC-B 实验分析

按照上述改进算法,对 MYGCC 的源代码进行了修改。然后用 MYGCC-B 重复第 1 节中的实验,实验输出结果中斜体部分是 MYGCC-B 相比原始 MYGCC 所增加的输出信息。将输出结果与源代码 test.c 进行对照,可以看到 MYGCC-B 正确找到并输出了控制流图中从"from"到"to"整条完整路径(例如:检查 test.c 时发现的第一个违反规则的路径 Line 5→Line 6→Line 13→Line 14)。与未改进前 MYGCC 的检查结果输出相对比,改进后的 MYGCC-B 丰富了自定义规则检查(Condate Checker)的输出信息,清晰地产生了错误生成的路径并展现给用户。通过直接观察错误生成的路径,错误可以被直观而迅速地发现,使得快速定位错误发生的场景成为可能。改进后的实验结果如下:

mygcc /root/test/test.c --tree-checks =/root/test/test.chk --tree-checks-verbose

/root/test/test.c: In function 'isleapyear':

/root/test/test.c:14: warning: user-defined warning malloc_deref: Unsafe dereference of X after malloc.

/root/test/test.c:14: instance = {X←p},

/root/test/test.c:14: reached: * p = 1.

Line 5: *p* = (*int* *) *D*.2136

Line 6: *D*.2137 = *year* % 400

Line 6: *if* (*D*.2137 == 0) *goto* <*L*0>; *else goto* <*L*1>;

<*L*1>:

Line 13: *year*.0 = (*unsigned int*) *year*

Line 13: *D*.2141 = *year*.0 & 3

Line 13: *if* (*D*.2141 != 0) *goto* <*L*4>; *else goto* <*L*2>;

<*L*2>:

Line 13: *D*.2142 = *year* % 100

Line 13: *if* (*D*.2142 == 0) *goto* <*L*4>; *else goto* <*L*3>;

<*L*3>:

Line 14: * *p* = 1

/root/test/test.c:21: warning: user-defined warning malloc_deref: Unsafe dereference of X after malloc.

/root/test/test.c:21: instance = {X←p},

/root/test/test.c:21: reached: D.2143 = * p.

Line 5: *p* = (*int* *) *D*.2136

Line 6: *D*.2137 = *year* % 400

Line 6: *if* (*D*.2137 == 0) *goto* <*L*0>; *else goto* <*L*1>;

<*L*1>:

Line 13: *year*.0 = (*unsigned int*) *year*

Line 13: *D*.2141 = *year*.0 & 3

Line 13: *if* (*D*.2141 != 0) *goto* <*L*4>; *else goto* <*L*2>;

<L4>:

Line 17: *if* (*p* ! = 0*B*) *goto* <*L5*>; *else goto* <*L6*>;

<L6>:

Line 21: *D*. 2143 = * *p*

/root/test/test. c: 9: warning: user-defined warning malloc_deref: Unsafe dereference of X after malloc.

/root/test/test. c:9: instance = {X←p},

/root/test/test. c:9: reached: * p = 1.

Line 5: *p* = (*int* *) *D*. 2136

Line 6: *D*. 2137 = *year* % 400

Line 6: *if* (*D*. 2137 == 0) *goto* <*L0*>; *else goto* <*L1*>;

<L0>:

Line 8: *printf* (&" %*d can be divided exactly by* 400 *n*"[0], *year*)

Line 9: * *p* = 1

5 结束语

本文提出了对 MYGCC 编程规则检查算法的一种改进方案,增加了输出违反检查规则的整条路径的功能。MYGCC 程序只能输出路径的起始点和终止点,而 MYGCC-B 可以输出违反待查性质的整条路径中每一个节点。此改进丰富了 MYGCC 的输出信息,有助于用户更迅速准确地找到错误发生的场景,方便错误的修正。

参考文献:

[1] Skolnik A. Fall 2007 Professor Aho Compilers and Interpreters Final Paper[EB/OL]. [2009-12-11]. http://gcc. gnu. org/wiki/Condate? action=AttachFile&do=view&target=Abe_Skolnik___Paper. pdf.

[2] Mygcc Overview[EB/OL]. [2009-12-20]. http://mygcc. free. fr/overview. html,2007.

[3] Examples in C of Using Mygcc[EB/OL]. [2009-12-20]. http://mygcc. free. fr/ex/c. html,2007.

[4] 陆赛因. 基于 MYGCC 的程序分析技术研究[D]. 长沙:国防科学技术大学,2008.

[5] Volanschi N. A Portable Compiler-Integrated Approach to Permanent Checking[J]. Automated Software Engineering, 2008,15(1):7-10.

[6] Volanschi E-N. Condate: A Proto-Language at the Confluence Between Checking and Compiling[C]// Proc of the 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06),2006:225-236.

李锋(1982 -),男,广东茂名人,硕士生,研究方向为程序静态分析技术。**E-mail**:li525235@263. net

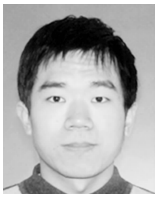
Li Feng, born in 1982, MS candidate, his research interest includes static analysis of programs.

文艳军(1975 -),男,湖北天门人,博士,副教授,研究方向为程序静态分析与验证。**E-mail**:yjwen@nudt. edu. cn

WEN Yan-jun, born in 1975, PhD, associate professor, his research interest includes static analysis and verification of programs.

齐治昌(1942 -),男,北京人,教授,博士生导师,研究方向为软件工程。**E-mail**:qzc@nudt. edu. cn

QI Zhi-chang, born in 1942, professor, PhD supervisor, his research interest includes software engineering.



陆赛因(1986 -),男,浙江湖州人,研究方向为程序静态分析技术。**E-mail**:science. lu@gmail. com

LU Sai-yin, born in 1986, his research interest includes static analysis of programs.