面向程序分析的插桩技术研究*

王克朝[1,2]，成　坚[1]，王甜甜[2]，任向民[1]
（1.哈尔滨学院 软件学院，哈尔滨 150086；2.哈尔滨工业大学 计算机科学与技术学院，哈尔滨 150001）

摘　要：为了满足测试覆盖分析和软件调试等程序分析技术对插桩技术的需求，提出了插桩模型，开发了一款实用的插桩工具。基于双缓冲技术，构建词法分析器和语法分析器。在语法分析归约时同步收集插桩信息，然后根据插桩策略执行插桩，生成目标文件。得到的程序运行时信息被应用于影响广泛的四种软件自动调试分析方法。对于这四种方法。缺陷语句均被准确识别为最可疑语句。应用结果表明，该方法能够为准确高效的程序分析提供必要的运行时信息。

关键词：程序插桩；程序分析；软件自动调试；语法树
中图分类号：TP311　　**文献标志码**：A　　**文章编号**：1001-3695(2015)02-0479-06
doi:10.3969/j.issn.1001-3695.2015.02.035

Program instrumentation oriented to program analysis

WANG Ke-chao[1,2], CHENG Jian[1], WANG Tian-tian[2], REN Xiang-min[1]
(1. *School of Software, Harbin University, Harbin* 150086, *China*; 2. *School of Computer Science & Technology, Harbin Institute of Technology, Harbin* 150001, *China*)

Abstract: In order to better meet the needs of program analysis technology such as test coverage analysis, and automatic debugging, this paper proposed a program instrumentation model and developed a practical tool. Based on double buffering technique, it constructed lexical analyzer and syntax analyzer. Instrumentation information was collected during syntactic deduction. Thus instrumentation action was performed and instrumented object file was generated according to instrumentation policy. It applied the program spectrum constructed by the instrumentation method to four state-art of coverage based software automatic debugging techniques. The defect statement was identified as the most suspicious statement by all the four techniques. It indicates that the program instrumentation method can provide necessary runtime information for accurate and efficient program analysis.

Key words: program instrumentation; program analysis; automatic software debugging; syntax tree

0　引言

随着软件系统越来越复杂，软件经常不像人们预期的那样运行，换句话说，软件并不总是可靠地运行，对计算机应用系统带来了不利影响，甚至造成了巨大的经济损失和灾难性的后果。因此，保证软件高可靠性已成为系统开发和维护工作的一个不可或缺的重要方面。

软件测试、分析和验证等技术被认为是提高软件质量和软件可靠性的重要手段，而程序插桩技术是程序分析和测试中的关键技术之一。

程序插桩[1~5]是在源程序的特定位置植入探针代码，并通过运行含有探针代码的程序来获取该程序运行时信息的技术。通过插桩可以收集程序在执行过程中某一时刻的运行状态，如基本块覆盖信息、谓词覆盖信息、路径覆盖信息、函数调用信息等。目前在测试覆盖分析[6]、测试用例自动生成[7]、测试用例约简[8]、软件性能分析与优化[9]、程序不变量分析[10]、软件自动调试[11~15]和软件缺陷检测与修复[16,17]等方面都得到了广泛的关注。

现有插桩工具大部分针对特定应用编制一个专用的插桩程序，但缺点是开发出的插桩工具难以满足通用应用程序。因此，迫切需要一款通用的插桩工具，以满足程序分析领域的需求。

本文提出了一个面向程序分析的插桩模型，并开发了一款通用的 C 源程序插桩工具。

1　面向程序分析的插桩模型

面向程序分析的插桩模型如图 1 所示。基于双缓冲机制，先对源代码进行插桩预处理，规范程序的编码规范；经过词法分析，将源程序识别为一个词法单元序列，且带有位置信息；经过语法分析，将词法单元序列进一步识别为不同的文法信息，构建插桩信息表；然后根据插桩策略和插桩信息表，执行其对应的插桩动作，生成插桩目标文件。

收稿日期：2014-02-17；**修回日期**：2014-04-08　　**基金项目**：哈尔滨学院青年基金资助项目(HUYF2014-007)；国家自然科学基金资助项目(61202092)；高等学校博士学科点专项科研基金资助项目(20112302120052)；黑龙江省教育规划青年专项课题；哈尔滨科技创新人才专项资金资助项目(RC2013QN010001，2014RFQXT062)；黑龙江省普通高等青年学术骨干项目；黑龙江省自然科学基金资助项目(F201127)；黑龙江省大学生创新创业资助项目

作者简介：王克朝(1980-)，男(蒙古族)，河南南阳人，讲师，博士研究生，主要研究方向为程序分析、软件调试(erickcwang@126.com)；成坚(1990-)，男，河南人，本科生，主要研究方向为程序分析；王甜甜(1980-)，女，辽宁丹东人，副教授，博士，主要研究方向为软件调试；任向民，男，哈尔滨人，教授，博士，主要研究方向为程序分析.

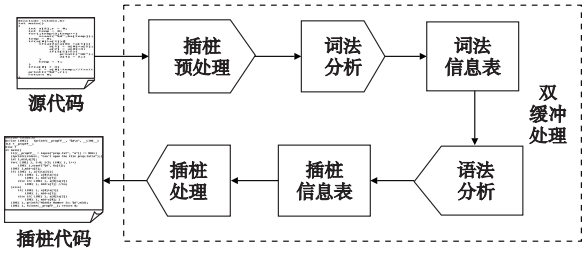

图1 面向程序分析的插桩模型

2 关键技术

2.1 双缓冲技术

插桩过程中需要多次读取程序源代码。每读取一个字符就需要访问一次磁盘,当源程序比较大时,效率显然是很低的。因此本文选择缓冲区技术,将文件的部分内容预先读到缓冲区中,当需要访问代码时,直接对这个缓冲区进行读取;当缓冲区中内容处理完毕后,再一次性地从磁盘中读取下一段内容到缓冲区。

1)双缓冲区

在插桩处理过程中,当前缓冲区的尾部很可能包含了当前识别单词的前一部分,而其另外一部分还在磁盘上。要获取整个单词就必须从磁盘上把下一段字符流读进来,但这种方式单词的前一部分就会被覆盖掉,造成对整个单词的识别失败。为了解决这一问题,本文引入了双缓冲机制。双缓冲区结构体信息如图 2 所示。

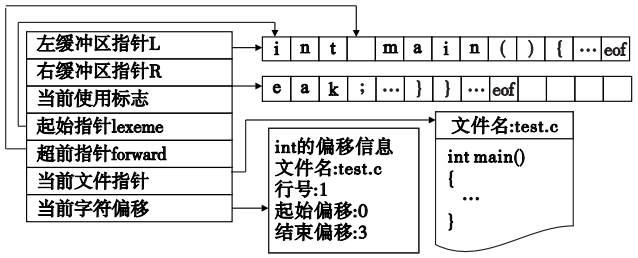

图2 双缓冲区结构体信息

设置两个读缓冲区,即左缓冲区 LEFT 和右缓冲区 RIGHT。每次只读取其中的一个,通过设置标志来记录当前使用的到底是哪一个缓冲区。假设每个缓冲区长度为 N,那么双缓冲区的长度为 $2N$。每次从磁盘读入 N 个字符,处理完毕后依据缓冲区使用标志,将源程序读到另一个缓冲区,同时当前使用标志指向新填充的缓冲区。如此循环,实现对整个源程序的处理。当剩余的源程序字符填不满整个缓冲区时,则在当前字符流最后一个字节处填入文件结束标志(EOF),这样在后续处理时,一旦遇到 EOF 就意味着源程序文件处理完毕。

对缓冲区的扫描通过使用两个指针来实现,即起始指针(lexeme)和超前指针(forward)。起始指针指向检索起始字符的地址,超前指针指向当前检索到的字符地址。考虑到 C 语言中的不对称边界原则(C 语言中采用把上界视做某序列中第一个被占用的元素,而把下界视做序列中第一个被释放的元素),起始指针和超前指针区间[lexeme, forward)中的字符流就是需要检索出的单词。

2)带标志的缓冲区

使用缓冲区时每次移动超前指针都必须检查是否到达了缓冲区的末尾,因而超前指针每向前一步,都需要判断到底是缓冲区末尾还是源文件结束,处理效率大大降低。

为了解决这个问题,本文采用延迟判断的策略。在每个缓冲区的末尾处各设置一个结束标志 EOF,这样每次移动超前指针时只需要检测是否是 EOF,从而实现将两次判断合并为对 EOF 的一次判断,而判断到底是缓冲区末尾还是文件结束,则延迟到第一次判断成功之后。

3)位置偏移信息

在插桩过程中,除了当前代码段的程序逻辑以外,还必须知道当前代码段的相对位置偏移,从而确定探针的插入位置,这对于提高插桩的准确性起着关键的作用。在读取缓冲区的同时获取和保存这些偏移信息。偏移信息包括标志符所在的文件名、行号、起始偏移(其开始位置相对于文件起始的偏移)、结束偏移(其结束位置相对于文件起始的偏移,也称超前偏移)。

初始状态起始指针和超前指针指向待扫描串的首字符,起始偏移和结束偏移也都指向首字符所在文件偏移。每次扫描超前指针向前移近,结束偏移也相应地增加,直到扫描出一个正确的串为止。整个串扫描过程中,开始指针与起始偏移同步,超前指针与结束偏移同步。这样,串扫描结束后,使用起始指针和超前指针就可以从缓冲区中取出当前串,而通过文件名、开始偏移和结束偏移就可以从源程序中取出当前串。因此,开始指针和超前指针及起始偏移和结束偏移,就将缓冲区中的单词串与源程序中的单词串对应起来,从而保证插桩过程中行号的一致性。

2.2 词法分析器

词法分析器将源程序中的字符串解析为有序的词法单词序列。为了更好地适应各种类型的源程序,同时方便后期语法分析和插桩处理,本文将词法单词存储在一个有序双向链表中。采用双向链表能够方便地在词法分析时插入词法单元、语法分析时顺序遍历词法单元、插桩时需要位置偏移等信息,因此存储的词法节点包括种别码(token kind)、属性值(data)、位置偏移(coord)三部分。

1)词法单元的识别策略

C 程序不同的词法单元往往是间隔存在的,相同类型词法单元之间被空格或其他型的词法单元隔开,如“int a,b;”,关键字 int 与标志符 a 之间被空格分隔,标志符 a 与 b 之间又被“,”运算符号分隔。根据这些特点,本文设计出一个词法分析器,首先将不同的词法单词进行分离,分出不同的层次,然后根据层次进行分析。

本文将 C 语言中的字符分成三类,即单词串型(如关键字、标志符)、间隔型(如运算符、分界符空格符和控制字符)、常量型(如字符常量、字符串常量以及数字常量)。在读取时,只能对单个字符进行处理。将组成这几类词法单词的字符单元提取出来,称为词法单词的词根。

在词法处理过程中,每识别到一个间隔型字符的词根,分别扫描前面串和后面串。前面串中很可能存在着一个没有检索出的串型单词,而后面串中可能存在着一个没有检索出来的运算符或者分界符。例如“i +=10”,当处理到字符“ +”,向前扫描将扫出标志符“i”,这正是串型单词,向后扫描扫出“ +=”,这正是间隔型单词。这个向前、向后扫描的过程称为 scanne 扫描。

2)超前搜索机制

采用超前搜索机制的词法分析过程如下:一旦检索到运算符或者分界符,就将当前串扫描出来,因为前面的串很可能是

一串型单词,然后依据识别标志符和关键字的状态转换图判断当前检索串到底是关键字还是标志符,接着对当前扫描出来的运算符或者分界符后向超前搜索,将提取出来的串按识别运算符和分界符的状态图进行识别。因此词法分析的一次扫描处理过程如下:

a)如果当前是分界符或运算符则执行 b),如果是数字则执行 d);

b)从当前位置开始向前进行 scanne 扫描,判断扫描出来的串到底是关键字还是标志符,将扫描的结果存入词法二元组链表;

c)从当前位置开始向后进行 scanne 扫描,判断扫描出来的串到底是运算符号还是分界符号,并将扫描的结果存入词法二元组链表,处理过程中如果待处理字符是'\''或'\"',则调用处理常量的函数进行处理,跳到 e);

d)依据浮点类型常量的状态转换图识别数字常量;

e)扫面完毕,准备进行下一次扫描。

2.3　语法分析器

考虑到插桩程序的可扩展性,本文采用自底向上的语法分析。一个自底向上的语法分析过程其实就是为源程序构造语法分析树的过程,在分析的同时构建插桩信息表。

1)LR 语法分析器

采用移近—归约自底向上的语法分析框架,包括输入缓冲区、输出缓冲区、分析栈和一个用于移近归约的语法分析表,其基本结构如图 3 所示。移近—归约框架同 LR 分析法协调配合完成语法分析,通过构造一个 SLR(1)型的语法分析表作为移近和归约的依据。

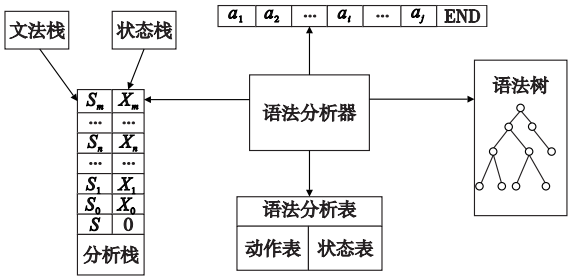

图3　语法分析器的基本结构

SLR(1)分析表包括动作表(action)和状态转移表(goto)。其中动作表指示当前状态下读到一个词法单元所需执行的移近—归约动作,而状态转移表则指示当一个归约完成后需要转移到的新的执行状态,因此 SLR(1)语法分析表的分析栈通常包括文法栈和状态栈。文法栈保存了当前移近的所有词法单词和归约出的文法信息,辅助构造语法分析树;状态栈存储语法分析表的状态信息,协同语法分析表指示下一步移近—归约的动作。

2)语法分析树的构建

语法树是在归约时同步建立的。假设当前归约的产生式右部长度为 n,则归约时文法栈中栈顶即将被弹出的 n 个元素之间是兄弟关系,而产生式左部数据为语法树的根,栈顶数据为根的最右孩子,距离栈顶第 $n-1(0\sim n-1)$ 个数据是根的最左孩子。

在构建语法树的过程中,由于产生式右部信息全部在栈中,只需建立它们之间的兄弟关系以及与产生式左部的父子关系即可。随着归约的不断进行,语法树不断被建立和合并,树的层次和规模不断壮大,直到接受词法序列的语法信息后,整个代码的语法分析树就会被建立。归约时同步建立语法树的算法如下所示。

算法　CreateSyntaxTree(s,S_i)

输入:分析栈的文法栈 s,归约的产生式 S_i。
输出:语法树的根 p。

```
1.    p 的数据域是 S_i 的左部的文法信息
2.    n 是 S_i 的右部长度
3.    P 的左右兄弟 leftBrother、rightBrother 的初始化为空
4.    P 的最右孩子 finalChild 的数据域是 s 的栈顶元素
5.    P 的最左孩子 firstChild 的数据域是 s 距离栈顶 n-1 的文法信息
6.    /*建立文法栈距离栈顶第 0 ~ 第 n-1 个元素间的兄弟关系*/
7.    for(i=0;i<n;i++)
8.    {
9.        P_i 是 s 距离栈顶第 i 个元素的信息
10.       P_i 的父亲节点是 p
11.       if(i==0)
12.           P_i 的 rightBrother 节点是空
13.       else{
14.           p_{i-1} 是 s 中距离栈顶第 i-1 个元素的信息
15.       P_i 的 rightBrother 节点是 p_{i-1}}
16.       if(i==n-1)
17.           P_i 的 leftBrother 节点为空
18.       else{
19.           P_{i+1} 是 s 中距离栈顶第 i+1 个元素的信息
20.           P_i 的 leftBrother 节点是 p_{i+1}}
21.   }
22.   return p;
```

3)分析栈的设计

SLR(1)语法分析栈的设计需要设计两个单独的栈,即文法栈和状态栈。前者的设计较简单,用一整型变量作为节点数据域即可,下面主要讨论文法栈的设计。

(1)文法栈节点数据域

设计文法节点,能同时存储词法节点单元和文法节点单元的结构体或者联合体,并通过一个控制标志来区分当前节点的数据域是词法数据还是文法数据。

(2)语法树节点与文法栈节点

在归约过程中构建语法树时,需将文法栈顶的几个节点弹出并存入语法树中,这样语法树节点和文法栈节点共用同一个结构体,即文法节点,如图 4 所示,其数据域是文法栈数据域节点。这样在构造语法树的过程中,只需直接将弹出的文法数据作为节点插入语法树。

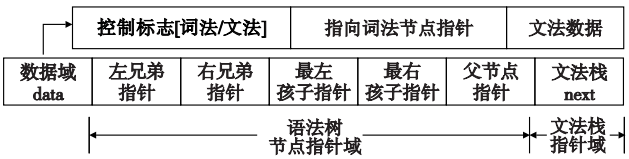

图4　文法节点结构

(3)文法节点的位置信息

为了能够识别出当前文法符号相对于源文件的位置,在文法节点中加入相对于源文件的位置信息。如果当前文法节点的数据域是词法节点,那么该节点的位置信息与数据域的位置信息相同;如果是文法节点,那么该节点的起始位置是其最左孩子的起始信息,终止位置就是其最右孩子的位置信息。这样在语法分析过程中将源文件中的代码段识别为符合 C 语法要求的文法序列,生成语法分析树,并将文法序列在源程序中的位置对应起来。

2.4　插桩器

1)插桩动作

插桩是在源程序的某些位置插入一些特定的信息。插桩

动作包括插桩位置和插桩信息。插桩位置是插桩点在源文件中的偏移,插桩信息是应插入的探针信息串。

插完桩的程序源代码编译运行后,会把探针信息的执行情况写入一个特定的文件中,这样通过分析这个特定文件就可以获取源程序的执行轨迹。而这个特定文件的创建、写入等操作都是通过探针来实现的,因此探针的主要功能就是创建文件、打开文件、关闭文件、写入文件等。创建文件可通过在源程序中插入一个文件信息全局变量来实现。打开/关闭文件需在主函数开始和结束的位置插入打开和关闭文件的功能代码。写文件是在源程序中写入代码执行的文件名和行号信息等功能代码,如 fprintf(fp, "%d %d\n", __FILE__, __LINE__)。

为了保证插桩的正确性,尤其不能改变源程序的代码逻辑,应针对不同程序在不同的地方插入探针。表达式记为 exp,探针函数记为 PPP。插桩动作如表 1 所示。

表 1 插桩动作

语句	文法描述	插桩后信息串
分支	if(exp)	if(PPP, exp)
	switch(exp)	switch(PPP, exp)
循环	while(exp)	while(PPP, exp)
	do while(exp);	do while (PPP, exp);
	for(exp;exp;exp)	for(PPP,exp;exp;PPP,exp)
其他	goto identifier;	PPP; goto identifier;
	continue;	PPP; continue;
	break;	PPP; break;
	return exp;	PPP; return exp;

2)插桩策略

插桩要求不能改变源程序的逻辑及行号。经过词法分析,源程序被识别为一个可被直接读取的词法单元序列,且带有位置信息。经过语法分析,词法单元序列被进一步识别为不同的文法信息,能够准确识别出到底是声明语句、循环语句,还是分支语句,同时也能识别出这些语句在源程序中的位置信息。随着文法单元的归约同步收集插桩信息,构建插桩信息表。语法分析完成后,根据插桩策略和插桩信息表,执行插桩动作,将探针写入源文件,生成插桩目标文件。出于效率考虑,可按照探针位置从小到大的顺序建立插桩信息表。源程序插桩算法如下所示:

算法 ProgramInstrumentation(ptable, src)

输入:双缓冲区 douBuf,处理文件指针 fp。
输出:插桩后源程序 dest。

```
1.   psrc 和 pdest 是分别指向 src 和 dest 的指针
2.   开始偏移 begOffset = 0
3.   结束偏移 endOffset = -1
4.   待写入字符数目 n = -1
5.   while(ptable! = NULL)
6.   {
7.       endOffset = ptable 的偏移数据域
8.       n = endOffset - begOffset
9.       将 psrc 中的 n 个字符写入 pdest
10.      将 ptable 的探针串写入 pdest
11.      ptable = patble. next
12.      begOffset = endOffset
13.  }
14.  return dest;
```

3 插桩实例

这里以从键盘输入三个数求中间值的程序段为例[11],如下所示,该代码中含有一处逻辑错误,在源程序中用"//bug"作了标志。

```
#include〈stdio. h〉
int main( )
{
    int i,mid,a[3];
    for(i=0;i<3;i++)
        scanf("%d", &a[i]);
    mid = a[2];
    if(a[1] < a[2]){
        if(a[0] < a[1])
            mid = a[1];
        else if(a[0] < a[2])
            mid = a[1];//bug
    }else{
        if(a[0] > a[1])
            mid = a[1];
        else if(a[0] > a[2])
            mid = a[0];}
    printf("Middle Number is: %d",mid);
    return 0;
}
```

经过本文的插桩工具插桩后的程序代码如下所示:

```
#include〈stdio. h〉
#define LINE( ) fprintf(__propFP__, "%d\n", __LINE__)
FILE * __propFP__;
#line 1
int main( )
{if((__propFP__ = fopen("prop. txt", "w")) == NULL){fprintf
(stderr, "can't open the file prop. txt\n");}
    int i,mid,a[3];
    for( LINE( ), i=0; i<3; LINE( ), i++)
        LINE( ),scanf("%d", &a[i]);
    LINE( ),mid = a[2];
    if( LINE( ), a[1] < a[2]){
        if( LINE( ), a[0] < a[1])
            LINE( ), mid = a[1];
        else if( LINE( ), a[0] < a[2])
            LINE( ), mid = a[1]; //bug
    }else{
        if( LINE( ), a[0] > a[1])
            LINE( ), mid = a[1];
        else if( LINE( ), a[0] > a[2])
            LINE( ), mid = a[0]; }
    LINE( ), printf("Middle Number is: %d",mid);
    LINE( ), fclose(__propFP__); return 0;
}
```

含有一处逻辑错误的源程序的测试用例如表 2 所示。

表 2 测试用例

测试用例编号	测试用例输入	期望输出
t_1	3,3,5	3
t_2	1,2,3	2
t_3	3,2,1	2
t_4	3,3,3	3
t_5	5,3,4	4
t_6	5,6,3	5
t_7	6,3,9	6

对插桩后的 C 程序进行编译、链接,生成可执行程序。运行生成的可执行程序时输入测试用例的输入,得到程序的执行结果和执行轨迹如表 3 所示,$t_1 \sim t_6$ 的实际输出结果与期望输出一致,则为成功测试用例;t_7 的不一致,则为失效测试用例。

生成的完整程序谱如表 4 所示,其中 P 表示成功测试用例,F 表示失效测试用例,1 表示测试用例执行轨迹经过了该语句实体,0 表示没有经过。

表 3　测试用例执行轨迹

编号	实际输出	执行轨迹
t_1	3	5 6 5 6 5 6 5 7 8 9 11 12 18 19
t_2	2	5 6 5 6 5 6 5 7 8 9 10 18 19
t_3	2	5 6 5 6 5 6 5 7 8 13 14 15 18 19
t_4	3	5 6 5 6 5 6 5 7 8 13 14 16 18 19
t_5	4	5 6 5 6 5 6 5 7 8 9 11 18 19
t_6	5	5 6 5 6 5 6 5 7 8 13 14 16 17 18 19
t_7	3	5 6 5 6 5 6 5 7 8 9 11 12 18 19

表 4　完整程序谱

程序源代码		测试用例						
S1	#include 〈stdio. h〉	t_1	t_2	t_3	t_4	t_5	t_6	t_7
S2	int main( )							
S3	{							
S4	int i,mid,a[3];	*P*	*P*	*P*	*P*	*P*	*P*	*F*
S5	for(i=0;i<3;i++)	1	1	1	1	1	1	1
S6	scanf("%d",&a[i])	1	1	1	1	1	1	1
S7	mid=a[2];	1	1	1	1	1	1	1
S8	if(a[1]<a[2]){	1	1	1	1	1	1	1
S9	if(a[0]<a[1])	1	1	0	0	1	0	1
S10	mid=a[1];	0	1	0	0	0	0	0
S11	else if(a[0]<a[2])	1	0	0	0	1	0	1
S12	mid=a[1];//bug	1	0	0	0	0	0	1
S13	}else{	0	0	1	1	0	1	0
S14	if(a[0]>a[1])	0	0	1	1	0	1	0
S15	mid=a[1];	0	0	1	0	0	0	0
S16	else if(a[0]>a[2])	0	0	0	1	0	1	0
S17	mid=a[0];}	0	0	0	0	0	1	0
S18	printf("Middle Number is:%d",mid);	1	1	1	1	1	1	1
S19	return 0;	1	1	1	1	1	1	1
S20	}							

4　插桩技术在程序分析中的应用

插桩技术在软件自动调试、测试用例自动生成、测试用例约简、软件测试覆盖分析(语句、谓词、路径等)、回归测试阶段用例优选、软件性能优化、提取不变量等程序分析领域得到了广泛的关注。本文以软件自动调试领域的应用为例来介绍插桩技术的应用。

目前在错误自动调试方法中覆盖分析法因具有计算复杂度低等优点得到了广泛的研究。其分析过程如下:

a)代码插桩,执行测试用例,收集执行轨迹。

b)统计分析失效用例和成功用例的执行覆盖信息,采用预先定义的计算公式计算各程序元素的可疑度,并将程序元素按可疑度由高到低排序,可疑度越高、排序越靠前,则越可能是缺陷语句。

覆盖分析法的主要思想是被越多失效用例以及越少成功用例执行的程序元素越可能含有缺陷。程序元素可以是语句、基本块、分支、函数等,本文基于语句覆盖。各种基于覆盖分析的自动调试方法都遵从上述步骤,区别在于采用的度量公式不同。Naish 等人[15]评价了 36 种采用不同度量公式的覆盖分析方法。证明尽管有些方法度量公式不同,但在可疑语句排序目的上是等价的。

表 5 列出了本文中使用的自动调试方法,选择这些方法的原因是:一方面,它们被经常用于软件自动调试分析相关实验中;另一方面,它们被 Naish 等人[15]证明对于 If-Then-Else-2 程序和多个方法等价,如 Tarantula 方法与 CBI 方法等价,Wong 方法与其他九个方法等价。

表 5　影响广泛的基于覆盖分析的软件自动调试技术

定位技术	语句可疑度度量公式	等价方法
Tarantula[12]	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	CBI
Jaccard[13]	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$	Anderberg, Sørensen-Dice, Dice, Goodman Rogers, Hamming, SimpleMatching, Hamann, Sokal, Euclid, M1, Manhattan, Lee
Wong[14]	$a_{ef}-a_{ep}$	
Optimal[15]	$\begin{cases}-1 & \text{if } a_{nf}>0\\ a_{np} & \text{otherwise}\end{cases}$	O^p

a_{ef}语句被测试用例集合中失效测试用例执行的次数

a_{nf}语句没有被测试用例集合中失效测试用例执行的次数

a_{ep}语句被测试用例集合中成功测试用例执行的次数

a_{np}语句没有被测试用例集合中成功测试用例执行的次数

基于表 5 中所列的四种软件自动调试分析方法和表 4 的完整程序谱,对含有一处逻辑错误的源程序代码进行自动调试的结果如表 6 所示。

从表 6 可以看出,语句 S12(mid=a[1];)在 Tarantula 调试方法中的可疑度为 0.86,排序第 1;在 Jaccard 调试方法中的可疑度为 0.50,排序第 1;在 Wong 调试方法中的可疑度为 0.00,排序第 1;在 Optimal 调试方法中的可疑度为 5.00,排名第 1。排名第 1 意味着是最可疑语句。这些应用结果表明,基于本文提出的插桩技术的软件自动调试方法能够比较准确地定位出逻辑错误位置,辅助开发人员进行软件调试,从而大大提高软件开发的效率。

5　与已有代码插桩工具的比较

本文开发的程序插桩工具与经典的代码插桩工具的比较分析如表 7 所示。已有工具如 codecover、emma、cobertura 等大多支持分析的程序设计语言为 Java,面向的应用大多为软件测试覆盖分析。而本文的程序插桩工具面向 C 程序,在语法分析归约时同步收集插桩信息,除了可以收集测试用例执行程序过程中的语句、分支、条件、循环等覆盖信息外,还可收集函数和程序执行的路径覆盖信息,因此不但可应用于软件测试覆盖分析、测试用例选择,还可应用于软件调试等过程中,为其提供充分的程序运行时信息。

6　结束语

本文提出一种面向程序分析的插桩方法,开发了一款适合程序分析的插桩工具。该方法基于双缓冲机制,通过词法分析将源程序识别为词法单元序列,并且通过位置信息将位置标志出来;通过语法分析将词法单元序列进一步识别为不同的文法信息,每归约出一个文法单元,就收集其插桩信息,构建插桩信息表;语法分析完成后,根据插桩策略和插桩信息表,执行相应的插桩动作完成程序插桩。在软件自动调试中的应用证明了本文插桩方法能够较好地满足程序分析对程序运行时信息的需求。

表6 基于插桩技术的软件自动调试结果

语句	测试用例							Tarantula		Jaccard		Wong		Optimal	
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	可疑度	排名	可疑度	排名	可疑度	排名	可疑度	排名
	P	*P*	*P*	*P*	*P*	*P*	*F*								
S5	1	1	1	1	1	1	1	0.5	4	0.14	4	-5.00	10	0.00	4
S6	1	1	1	1	1	1	1	0.5	4	0.14	4	-5.00	10	0.00	4
S7	1	1	1	1	1	1	1	0.5	4	0.14	4	-5.00	10	0.00	4
S8	1	1	1	1	1	1	1	0.5	4	0.14	4	-5.00	10	0.00	4
S9	1	1	0	0	1	0	1	0.67	3	0.25	3	-2.00	6	3.00	3
S10	0	1	0	0	0	0	0	0	10	0.00	10	-1.00	2	-1.00	10
S11	1	0	0	0	1	0	1	0.75	2	0.33	2	-1.00	2	4.00	2
S12	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0.86**	**1**	**0.50**	**1**	**0.00**	**1**	**5.00**	**1**
S13	0	0	1	1	0	1	0	0	10	0.00	10	-3.00	8	-1.00	10
S14	0	0	1	1	0	1	0	0	10	0.00	10	-3.00	8	-1.00	10
S15	0	0	1	0	0	0	0	0	10	0.00	10	-1.00	2	-1.00	10
S16	0	0	0	1	0	1	0	0	10	0.00	10	-2.00	6	-1.00	10
S17	0	0	0	0	0	1	0	0	10	0.00	10	-1.00	2	-1.00	10
S18	1	1	1	1	1	1	1	0.5	4	0.14	4	-5.00	10	0.00	4
S19	1	1	1	1	1	1	1	0.5	4	0.14	4	-5.00	10	0.00	4

表7 本文工具与经典的代码插桩工具的比较

工具	支持的程序语言	支持的覆盖分析类型	面向的应用
codecover http://codecover.org/	Java、COBOL	语句、分支、条件、循环	软件测试的覆盖分析、测试用例选择
emma http://emma.sourceforge.net/	Java	类、方法、代码行、基本块	软件测试的覆盖分析
cobertura http://cobertura.github.io/cobertura	Java	语句	软件测试的覆盖分析
本文工具	C	语句、分支、条件、循环、函数、路径	软件测试覆盖分析、测试用例选择、软件调试等

参考文献:

[1] 郑晓梅.一个基于Eclipse的通用Java程序插桩工具[J].计算机科学,2011,38(7):139-143.

[2] 晏华,尹立孟.代码自动插桩技术的研究与实现[J].电子科技大学学报,2002,31(1):62-66.

[3] 李业华,顾乃杰,张颖楠,等.基于插桩和布尔逻辑的运行时程序验证框架[J].计算机工程,2013,39(1):29-34.

[4] 王者思,叶东升,张建伟.软件测试中评价函数的构造以及插桩方法的研究[J].计算机工程与设计,2013,34(5):1673-1680.

[5] CHITTIMALLI P K,SHAH V. GEMS:a generic model based source code instrumentation framework[C]//Proc of the 5th International Conference on Software Testing,Verification and Validation. Washington DC:IEEE Computer Society,2012:909-914.

[6] 刘慧梅,徐华宇.软件测试中代码分析与插桩技术的研究[J].计算机工程,2007,33(1):86-88.

[7] MARAGATHAVALLI P, KANMANI S, KIRUBAKAR J S,*et al*. Automatic program instrumentation in generation of test data using genetic algorithm for multiple paths coverage[C]//Proc of IEEE International Conference on Advances in Engineering, Science and Management. Washington DC:IEEE Press,2012:349-353.

[8] GONG Dan-dan, WANG Tian-tian, SU Xiao-hong, *et al*. A test-suite reduction approach to improving fault-localization effectiveness[J]. Computer Languages, Systems & Structures,2013,39(3):95-108.

[9] 马桂杰,蒋昌俊,刘吟,等.基于插桩技术的并行程序性能分析方法设计和实现[J].计算机应用研究,2007,24(10):225-228.

[10] HANGAL S,LAM M S. Tracking down software bugs using automatic anomaly detection[C]//Proc of the 24th International Conference on Software Engineering. New York:ACM Press,2002:291-301.

[11] YU Yan-bin,JONES J A,HARROLD M J. An empirical study of the effects of test-suite reduction on fault localization[C]//Proc of the 30th ACM International Conference on Software Engineering. New York:ACM Press,2008:201-210.

[12] JONES J A, HARROLD M J, STASKO J. Visualization of test information to assist fault localization[C]//Proc of the 24th International Conference on Software Engineering. New York:ACM Press,2002:467-477.

[13] ABREU R,ZOETEWEIJ P,Van GEMUND A J C. On the accuracy of spectrum-based fault localization[C]//Proc of the Testing:Academic and Industrial Conference on Practice and Research Techniques—MUTATION. Washington DC:IEEE Computer Society,2007:89-98.

[14] WONG W E,QI Yu,ZHAO Lei,*et al*. Effective fault localization using code coverage[C]//Proc of the 31st Annual International Conference on Computer Software and Applications. Washington DC:IEEE Computer Society, 2007:449-456.

[15] NAISH L, LEE H J, RANAMOHANARAO K. A model for spectra-based software diagnosis[J]. ACM Trans on Software Engineering and Methodology,2011,20(3):11:1-11:32.

[16] Le GOUES C, DEWEY-VOGT M, FORREST S, *et al*. A systematic study of automated program repair: fixing 55 out of 105 bugs for MYM8 each[C]//Proc of the 34th International Conference on Software Engineering. Washington DC:IEEE Press,2012:3-13.

[17] NGUYEN H D T, QI Da-wei,ROYCHOUDHURY A,*et al*. SemFix:program repair via semantic analysis[C]//Proc of International Conference on Software Engineering. Washington DC:IEEE Press,2013:772-781.