第 35 卷 第 7 期 计 算 机 工 程 2009 年 4 月
Vol.35 No.7 Computer Engineering April 2009
·软件技术与数据库· 文章编号：1000—3428(2009)07—0039—03 文献标识码：A 中图分类号：TP311

基于 LCC 的测试程序控制流路径子集生成算法

陈 宇[1]，李志蜀[1]，金 虎[1,2]，何 江[1]
(1. 四川大学计算机学院，成都 610064；2. 成都信息工程学院计算机系，成都 610041)

摘 要：针对路径覆盖测试技术中如何计算被测试程序的有效控制流路径子集的关键性问题，提出一种利用 LCC 编译器的前端结果来生成基于一次循环策略的测试程序控制流路径子集的算法。该算法通过引入邻接矩阵并借助自定义的堆栈数据结构来完成控制流路径子集的生成。通过实验程序对算法进行检验，结果表明，该方法能高效准确地计算出待测源程序片的控制流路径子集。

关键词：LCC 编译器；控制流路径；邻接矩阵

Control Flow Paths Subset of Tested Program Generation Algorithm Based on LCC

CHEN Yu[1], LI Zhi-shu[1], JIN Hu[1,2], HE Jiang[1]
(1. School of Computer, Sichuan University, Chengdu 610064;
2. Department of Computer Science, Chengdu University of Info. Tech., Chengdu 610041)

【Abstract】To solve the key problem of how to calculate the effective subset of the control flow paths of the tested program, this paper proposes an algorithm for working out the subset of the whole control flow paths of the program with the help of the LCC compiler in software coverage testing. The control flow paths in the subset are based on the strategy that the loop statements in the program execute less than twice. Adjoining matrix is introduced by the algorithm to work out the subset of the control flow paths with the help of customized stack structure. The correctness of the algorithm is verified on a sample program, which indicates that the subset of the whole control flow paths of the tested program can be calculated efficiently and accurately with this method.

【Key words】LCC compiler; control flow path; adjoining matrix

1 概述

覆盖测试是一种白盒测试技术，通过运行被测程序，计算得到程序各类语句执行的覆盖率，并对代码的执行路径覆盖范围进行评估、分析，以帮助测试者找出被测程序中的错误。作为针对结构测试的一种常用的充分性准则，路径覆盖要求设计足够的测试用例使得被测程序的每条可能路径都至少执行一次。但实际上，即使是规模很小的程序，包含的逻辑路径数量也是相当大的，从而使得完全的路径覆盖在实际的软件测试中并不具有很大的可行性[1]。因此，路径测试要求能够找出被测程序中的所有逻辑路径的有效路径子集，以便通过设计可覆盖该有效逻辑路径子集的用例来发现程序中尽可能多的错误[2]。本文提出了一种利用 LCC 编译器的前端结果来生成基于一次循环策略的测试程序控制流路径子集的算法。

2 LCC编译器简介

LCC 编译器是一个具有产品级质量的用于研究的 C 编译器，在 Unix 界广为流行。从 1988 年开始，LCC 开始用于编译实际程序，因为 LCC 开放源代码的特点，以及对多种机器平台的良好支持，使其无论在程序开发还是科学研究中都得到广泛的应用。

LCC 使用无环路有向图(Directed Acyclic Graph, DAG)对中间代码进行描述，它使用二叉树的链表形式进行组织，所有的中间代码通过代码表进行管理[3]。每个代码表就是 DAG 组成的序列或森林，DAG 节点的定义如下[3]：

```
typedef struct node *Node;
struct node {
    short op;
//DAG 操作符，用整数形式标示该节点执行的语义操作
    short count;
//记录节点的值被使用或被其他节点引用的次数
    Symbol syms[3];  //部分操作符使用符号表指针
//作为操作数，这些操作数存放在 syms 中
    Node kids[2];    //指向操作数节点
    Node link;       //指向森林中下一个 DAG 的根
    Xnode x;         //x 域是后端对节点的扩展，后
//端使用其来存放生成代码时需要的各节点的数据
};
```

LCC 的代码表及 DAG 节点作为算法的核心操作对象，其结构定义决定了算法的复杂性与遍历方式，算法的工作本质上即通过访问并操作该数据结构以最终得到程序的分支路径子集。LCC 编译器前端将源代码语句编译为语法和语义分析树然后转换为由上述节点结构组成的 DAG 结构。并通过每条语句的 DAG 结构根节点的 link 域将每条语句的 DAG 依次链接成单链表的形式，从而形成 DAG 序列或森林，并最

基金项目：四川省科技攻关基金资助项目“软件自动测试技术研究与自动测试工具开发”(05GG021-003-2)；科技部科技型中小企业技术创新基金资助项目“软件自动测试工具 WsSTKit”(0626225101730)
作者简介：陈 宇(1983－)，男，硕士，主研方向：计算机网络，信息系统；李志蜀，教授；金 虎，博士；何 江，硕士
收稿日期：2008-09-10 **E-mail：**chuanlao2002@163.com

后将序列或森林添加到代码表中[3]。这样，通过遍历代码表和 DAG 森林便可以到达每条源代码语句的 DAG 结构，通过判断其 DAG 节点的 op 域操作符便可获得每条源语句的语义操作信息和控制流信息。

3 控制流路径子集的生成

算法的基本思路是:根据 LCC 前端生成的代码表及 DAG 结构，构造表示程序控制流图的邻接矩阵。遍历邻接矩阵，生成基于一次循环策略的程序控制流路径子集。

整个算法的数据流图(Data Flow Diagram, DFD)如图 1 所示，其中，由邻接矩阵生成程序控制流路径子集的过程是算法的核心部分。

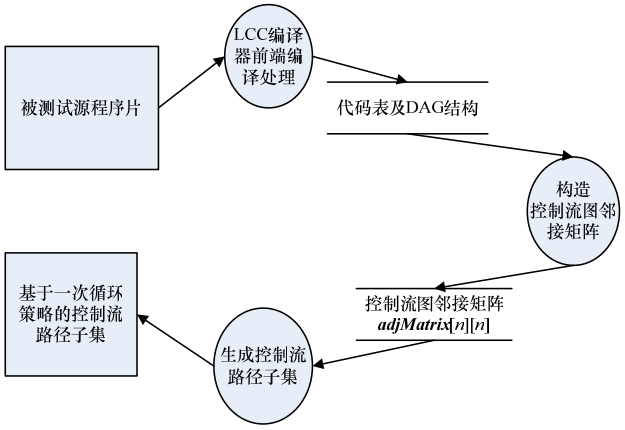

图 1　算法的整体数据流图

3.1 构造邻接矩阵

含有 n 条语句的待测源代码语句片的邻接矩阵是一个 $n \times n$ 的矩阵，可用二维整型数组 int adjMatrix[n][n]表示。其中，当且仅当语句 i 和语句 j 之间有可以直接执行到达的控制流时，元素 adjMatrix[i][j]被赋值为 1，否则赋值为 0[4]。邻接矩阵的引入使得由 DAG 生成控制流路径子集的过程得以简化，其既保留了 DAG 中包含的用于控制流路径子集生成的足够信息，又简化了遍历对象的数据结构的复杂性。

算法 1 实现了由 LCC 前端生成的 DAG 结构生成程序流图邻接矩阵的计算过程。由于 LCC 前端编译被测试源程序所生成的代码表就是由 DAG 组成的序列(森林)，而序列中的每个 DAG 与被测试代码的源语句存在一一对应的关系，因此，通过访问与语句 i 对应的 DAG 的根节点 op 域的值便可获得该语句执行时的静态控制流信息。

算法 1 genAdjMatrix

输入：LCC 前端编译被测程序 G 生成的 DAG 森林 DAG_F

输出：包含程序 G 控制流信息的邻接矩阵 ***adjMatrix***[n][n]

变量：当前处理语句的语句号 i

STEP 1 将变量 i 初始化为程序 G 的首条语句的源代码语句号。

STEP 2 若语句 i 为 G 的最后一条语句，则算法结束。否则执行 STEP 3。

STEP 3 若语句 i 具有条件跳转语义(getNodeOp(DAG_F, i)值为 EQ, GE, GT, LE, LT, NE)，则调用条件跳转语句处理函数 ConJmp(DAG_F,i)对语句 i 进行处理，同时流程跳转到 STEP 6；否则执行 STEP 4。

STEP 4 若语句 i 具有无条件跳转语义(getNodeOp(DAG_F,i)值为 JUMP、CALL)，则调用 JumpTo(DAG_F,i)函数计算语句 i 跳转到语句的语句号 j，adjMatrix[i][j]=1，同时流程跳转到 STEP 6；否则执行 STEP 5。

STEP 5 adjMatrix[i][i+1]=1。

STEP 6 i=i+1,处理下一条语句；重复执行 STEP 2。

END genAdjMatrix

算法 1 中存在对算法 2——条件跳转语句处理函数的调用。算法 2 对具有条件跳转语义的语句 i 进行处理，其算法内部包含对自身的递归调用，用于处理被测试源程序片中紧接着条件跳转语句的条件跳转语句。

算法 2 ConJmp

输入：LCC 前端编译被测程序 G 生成的 DAG 森林 DAG_F，当前处理语句的语句号 i

输出：邻接矩阵 ***adjMatrix*** 的特定元素的值

STEP 1 若语句 i+1 具有条件跳转语义(getNodeOp(DAG_F,i+1)值为 EQ, GE, GT, LE, LT, NE)，则递归调用条件跳转语句处理函数 ConJmp(DAG_F,i+1)对语句 i+1 进行处理，同时流程跳转到 STEP 4；否则执行 STEP 2。

STEP 2 若语句 i+1 具有无条件跳转语义(getNodeOp(DAG_F,i+1)值为 JUMP, CALL)，则调用 JumpTo(DAG_F,i+1)函数计算语句 i+1 所跳转到语句的语句号 j，adjMatrix[i+1][j]=1，同时，流程跳转到 STEP 4；否则执行 STEP 3。

STEP3 若语句 i+1 不是 G 的最后一条语句，则 adjMatrix[i+1][i+2]=1。

STEP 4 调用 JumpTo(i)函数计算语句 i 跳转到语句的语句号 j，adjMatrix[i][j]=1。

END ConJmp

这样通过算法 genAdjMatrix 的处理，整个测试程序分支路径子集生成算法便完成了根据 LCC 前端生成的代码表及 DAG 结构生成被测程序的控制流图邻接矩阵的子处理过程。

3.2 邻接矩阵生成控制流路径子集

算法 genCFP 采用深度优先多次回溯的方式遍历邻接矩阵，以模拟程序控制流的执行，并通过跟踪和记录控制流执行到的语句节点得到基于一次循环策略的所有程序控制流路径。为保存可供回溯的路径节点信息，需要借助一个外部定义的堆栈数据结构，其中堆栈中的每个节点就代表了程序的控制流路径节点。同时为了避免回溯过程中的重复遍历，堆栈节点保留了其他额外的重要信息。堆栈节点定义如下:

```
class CStackNode
{
public:
  int m_nLoopPos;
//保存深度遍历时可供回溯的上一层节点的位置
  bool m_bHasChild;        //标识此堆栈节点是否存在子节点
  int m_nNodeID;           //当前堆栈节点的源代码语句号
};
```

而堆栈则设计为由该堆栈节点类组成的链式栈。同时，为满足程序中循环结构的循环体部分或者不执行或者只执行一次的策略，算法需要一个与邻接矩阵相同大小的标记矩阵，用于记录因为程序的循环结构而出现语句逆向执行的情况。如此，算法便可通过标记矩阵获得深度遍历时所模拟的控制流执行到的循环体中语句节点的信息。通过限制循环体内语句被控制流执行到的次数不大于 1，便可生成满足程序中循环结构或者不执行或者只执行一次策略的所有控制流路径。

算法 3 genCFP

输入：被测程序 G 的控制流邻接矩阵 ***adjMatrix***[n][n]

输出：基于一次循环策略的程序控制流路径子集 CFP_S

变量：保存可供回溯的路径节点的堆栈 CStack，标记矩阵 ***flagMatrix***[n][n]，堆栈节点 I, j，算法生成的一条控制流路径 CFP_I

STEP 1 初始化 CFP_S, CStack 与 flagMatrix。将表示被测程序 G 首条语句的堆栈节点压入 CStack。

STEP 2 若 CStack 为空则算法结束，否则执行 STEP 3。

STEP3 获取 CStack 的栈顶节点并赋给 i，调用 nextAdjNodeNotReach(adjMatrix,i)函数获得节点 i 的下一个未深度遍历过的邻接节点并赋给 j，执行 STEP 4；若 i 无邻接节点或其邻接节点都已遍历过，则流程跳转到 STEP 7。

STEP 4 若 i.m_nNodeID > j.m_nNodeID，表明此为循环语句中控制流的逆向执行过程，算法执行 STEP 5，否则执行 STEP 6。

STEP 5 若 flagMatrix[i][j]==1，表明该逆向执行已经执行过，则流程跳转到 STEP 3 重复执行，否则执行 STEP 6。

STEP 6 节点 j 入栈 CStack。

STEP 7 节点 i 弹栈；若 i.m_bHasChild == false，执行 STEP 8，否则重复执行 STEP 2。

STEP 8 CStack 中的现有节点便组成一条控制流路径，将其依次输出到 CFP_I，并将 CFP_I 添加到 CFP_S 中；重复执行 STEP 2。

END genCFP

通过算法 genCFP 的处理，被测试程序的满足一次循环策略的所有程序控制流路径便全部生成并保存在 CFP_S 中。

4 算法时间复杂度分析

整个算法过程由邻接矩阵生成子过程和控制流路径生成子过程组成。则整个算法的时间取决于两者的执行时间之和。

算法的邻接矩阵生成子过程的关键操作在于根据 LCC 前端生成的代码表(DAG 森林)中 DAG 的根节点的 op 域的不同值调用不同处理函数，从而最终得到该(语句)节点的下一条邻接(语句)节点的源语句号。其时间渐进复杂度主要由待测源程序片断中跳转结构的复杂性决定。由于 LCC 编译器前端已经对死的跳转指令进行了处理，使得 DAG 结构中不会出现跳转到跳转指令节点的跳转指令节点和跳转到紧接着的标号的跳转指令节点[3]，这样就将邻接矩阵生成子过程的时间渐进复杂度控制在多项式级，在最坏情况下，即出现由源程序首语句到尾语句的跳转语句情况下，子过程的时间复杂度 $T(N)=O(N^2)$。

算法的控制流路径子集生成子过程的关键操作在于通过遍历邻接矩阵的方式来模拟程序控制流的执行，以跟踪并记录下控制流执行到的语句节点。由于算法借助一个堆栈数据结构来保存可供回溯的路径节点，因此算法的渐进时间复杂度就主要由堆栈中元素的最大数目决定。即深度遍历时的最大深度(等价于最长控制流路径的长度)。设其值为 K，源语句节点数为 M，由于限定了程序中循环结构或者不执行或者只执行一次的策略，因此易知 $K< 2M$。

判断堆栈不空的 while 循环中嵌套了两层循环，而两层循环均是对邻接矩阵第 i 行向量的遍历。故根据时间渐进复杂度大 O 表示法的乘法规则，在最坏情况下，该子过程的时间复杂度 $T(M)=O(K\times M^2)$。其中，K 为待测源程序中存在的最长控制流路径长度。其与待测源程序的实际结构密切相关。

综上所述，根据大 O 表示法的加法规则[5]：

$$T(n,m)=T_1(n)+T_2(m)=O(\max(f(n),g(n)))$$

可得整个算法的时间渐进复杂度为

$$O(\max(T(N),T(M)))=O(\max(O(N^2),O(K\times M^2)))$$

5 实验结果

为了检验和评估上文所述算法的效果，这里选用一个用 ANSI C 编写的简单源程序片段进行实验，由于算法本身的目的在于生成源程序的控制流路径，因此这里选用了一段逻辑密集、计算稀疏[6]的程序片断(图 2、图 3)，以便于产生更好的实验效果。左边序号(1~10)为源代码语句的编号。

```
int arithmeticst_validation ()
{
        int x1,x11,x2,x22,x3,x33;
1       if(x1 > 0)
2               x11 = 1;
3       else if(x1 == 0)
4               x11 = 0;
5       else    x11 = -1;
6       while(x2 > 0)
7               x22 ++ ;
8   do          x33++;
9   while(x3>0);
10   return 0;}
```

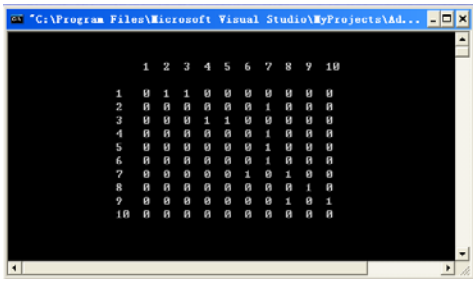

图 2 被验证程序的邻接矩阵

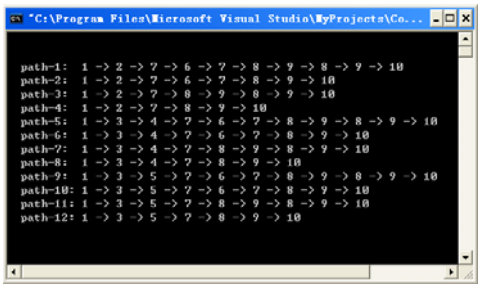

图 3 生成的控制流路径子集

6 结束语

本文讨论一种利用 LCC 编译器前端所生成的中间代码结构来生成被测试程序控制流路径子集的算法，通过使用邻接矩阵来处理控制流信息及生成控制流路径，并通过一个实验程序对算法的效果进行了检验和评估。实验表明，该方法能高效准确地计算出待测源程序片的控制流路径子集，为路径覆盖测试的开展奠定了基础。下一步的工作是研究和实现控制流路径生成算法对于不同测试策略的支持以及研究如何根据路径子集有针对性地设计测试用例的技术。

参考文献

[1] Myers G J. The Art of Software Testing[M]. New York, USA: John Wiley & Sons, Inc., 1979.

[2] Bertolino A, Marré M. How Many Paths are Needed for Branch Testing?[J]. Journal of Systems and Software, 1996, 35(2): 95-106.

[3] Fraser C W, Hanson D R. A Retargetable C Compiler Design and Implementation[M]. 北京: 电子工业出版社, 2005.

[4] Jorgensen P C. Software Testing[M]. 北京: 机械工业出版社, 2003.

[5] 殷人昆. 数据结构: 用面向对象方法与 C++描述[M]. 北京: 清华大学出版社, 1999.

[6] Huang J C. Program Instrumentation and Software Testing[J]. Computer, 1978, 11(4): 3-8.

编辑 索书志