

路径测试中基本路径集自动生成方法的研究

韩 寒, 姜淑娟

(中国矿业大学 计算机科学与技术学院, 江苏 徐州 221116)

摘 要: 本文提出了一种有效、自动的基本路径集的生成方法, 它能在一定程度上提高软件测试的效率. 该方法首先设计搜索算法生成基本路径集; 再根据基本块之间的依赖性分析来识别不可达路径并将其删除; 然后向路径集中添加未覆盖边所在的可达独立路径; 最后向路径集中添加未找到可达路径的未覆盖边所在的不可达独立路径, 及线性运算所需的不可达独立路径. 实验结果表明, 该基本路径集生成方法能有效地生成包含尽可能多的可达路径的基本路径集.

关键词: 基本路径集; 不可达路径; 可达路径; 依赖性分析

中图分类号: TP311

文献标识码: A

文章编号: 1000-7180(2013)01-0104-06

A Research on Automatic Generation of Basis Path Set in Path Test

HAN Han, JIANG Shu-juan

(School of Computer Science and Technology, China University of
Mining and Technology, Xuzhou 221116, China)

Abstract: This paper provides an effective and automatic method of generating basis path set, which is significant in improving the efficiency of software testing. Firstly, a basis path set of the program is generated by visiting all nodes of the program's control flow graph; secondly, infeasible paths are detected according to the dependent analysis among blocks and are deleted from the basis path set at the same time; then, some feasible independent paths which contain the edges which are not covered by the path set are added into the path set; finally, some infeasible independent paths which contain the edges which are not covered by the path set or are needed by linear operations. The author applies the proposed method in some typical programs, and the result shows that the proposed method can generate a basis path set which contains as many feasible paths as possible efficiently.

Key words: basis path set; infeasible path; feasible path; dependency analysis

1 引言

软件测试包括功能测试和结构测试, 其中结构测试针对程序的内部结构进行测试, 其目的是充分覆盖整个软件系统, 检验程序中的每条通路是否都能按预定要求正确工作^[1]. 然而在实际软件系统中, 即使是一个不太复杂的程序, 其路径的组合都有可能是一个庞大的数字, 因此, 对程序中的所有路径进行测试需要巨大的资源^[2]. 基本路径测试^[2]较好地解决了这一难题, 它通过找到一组合理的路径, 使这

组路径分别对应一个测试用例, 以达到基路径覆盖的目的. 目前有代表性的基本路径集生成方法^[3-6]尚需改进和完善.

本文提出了一种基本路径集的生成方法, 该方法在保证最终生成的路径集是基本路径集的前提下, 能包含尽可能多的可达独立路径.

2 基本概念

圈复杂度: 又称 McCabe 复杂性度量, 简单的定义为控制流图的区域数. 从圈复杂度可导出基本路

收稿日期: 2012-05-29; 修回日期: 2012-06-15

基金项目: 国家自然科学基金(60970032); 江苏省自然科学基金(BK2008124); 江苏省研究生培养创新工程(CXLX12_0950)

径集中的独立路径数.通常圈复杂度可用以下三种方法求得:

①将圈复杂度定义为控制流图中的区域数.

②设 E 为控制流图的边数, N 为图的节点数,则定义圈复杂度为 $V(G)=E-N+2$.

③若设 P 为控制流图中的判定节点数,则有 $V(G)=P+1$.

独立路径^[7]:至少包含一条在其它路径中未有过边的路径.

基本路径集^[1]:基本路径集是程序的部分路径集合,其满足如下性质:①每条路径都至少包含一条其它路径未包含的边;②程序中所有的边都被基本路径集中的路径访问;③程序中的所有的、不属于基本路径集的路径都可由基本路径集中的路径经过线性运算得到.

数据依赖^[8]:设 m, n 为控制流图中的两个节点, v 为一变量,若下列3个条件满足,则称 m 关于变量 v 数据依赖于 n ,记为 $DD(m, n, v)$,也可记为 $DD(m, n)$.

① n 对变量 v 进行了定义,即 $v \in \text{def}[n]$;

② m 执行时使用了 v 的值,即 $v \in \text{use}[m]$;

③ n 与 m 之间存在一条可执行路径,并且在此路径上,没有语句对 v 重新定义.

控制依赖^[8]:设 m, n 为控制流图中的两个节点,若下列3个条件满足,则 n 控制依赖 m ,记为 $CD(n, m)$.

①从 m 到 n 之间存在一条可执行路径 P ;

②对 P 上除 m, n 外的每个节点 n' ,节点 n 都是它的后必经节点;

③节点 n 不是节点 m 的后必经节点.

3 基本路径集生成方法

本文提出了一种基本路径集的生成方法,该方法的总体设计思想如下:

(1)搜索基本路径集:设计搜索算法,对程序控制流图进行遍历生成基本路径集;

(2)确定并删除不可达路径:对路径集中的路径进行依赖性分析,识别不可达路径并将其删除;

(3)添加可达独立路径:向路径集中添加未覆盖边所在的可达独立路径;

(4)添加不可达独立路径:向路径集中添加未找到可达路径的未覆盖边所在的不可达独立路径,及线性运算所需的不可达独立路径.

本文利用 Java 分析优化工具框架 Soot^[9] 的编

译分析功能,对源程序扫描后构造一棵等价于源程序的抽象语法树,然后通过遍历语法树,对各语句进行处理,产生中间表示,最后将中间表示序列划分为基本块并建立控制流图.

3.1 基本路径集搜索算法

用一个链表数组 $bPaths$ 来存储基本路径集,该链表数组的大小为程序控制流图的圈复杂度.对于程序中出现的循环,本文仅考虑循环 0 次和 1 次的情况.本步搜索算法如算法 1 所示.

算法 1:基本路径集搜索算法

```

输入:程序的 CFG
输出:程序的基本路径集 bPaths
Begin
1: int pathFlag=1; //路径复制变量
2: bPaths.length()=V(G); //链表数组的大小为程序的圈复杂度
3: for(each path∈bPaths) do
4:   当前节点 curNode 初始化;
5:   if(CFG.getTails().contains(curNode)) then //判断是否为出口节点
6:     path.add(curNode);
7:     break;
8:   else if(outdegree(curNode)≥2) then //判断出度是否≥2
9:     if(curNode 已经分析过) then
10:      取其它后继;
11:     endif
12:     if(path 已两次经过同一节点) then
13:      若该节点的出度为 m,将当前路径中 curNode 及其之前的节点复制到 bPaths[pathFlag]~bPaths[pathFlag+m-2];
14:      pathFlag=pathFlag+m-2;
15:     endif
16:     if(indegree(curNode)≥2) then //判断是否为多人多出节点
17:      if(path 中 curNode 前一节点为 curNode 的第一个前驱) then
18:      若该节点的出度为 m,将当前路径中 curNode 及其之前的节点复制到 bPaths[pathFlag]~bPaths[pathFlag+m-2];
19:      pathFlag=pathFlag+m-2;
20:     endif
21:     取第一个后继;
22:   endif
23: else 取 curNode 的唯一后继节点;
24: endif
25: endif
26: endfor
End

```

算法 1 中,不论是循环语句还是选择语句(包括 if 语句和 switch 语句),其所在节点都可归类为出

度 ≥ 2 的节点(行 8). 当遇到一个出度 ≥ 2 的节点 n 时, 若 n 的出度为 m , 则当前路径继续记录取第一个后继节点的路径, 将当前路径中 n 及其之前的节点复制到 $bPaths[pathFlag] \sim bPaths[pathFlag+m-2]$ 中 ($pathFlag$ 的初值为 1), 然后用 $bPaths[pathFlag] \sim bPaths[pathFlag+m-2]$ 分别记录取其它后继节点的路径, 复制完后令 $pathFlag = pathFlag + m - 2$ (行 12-20). 记录当前路径直到程序结束, 检查 $bPaths$ 中是否有未完成的路径, 如果有未完成的路径则用与前面相同方法进行路径记录, 当路径全部完成后输出路径.

对于圈复杂度的计算, 本文提出了一种新的计算方法, 若控制流图 G 的节点数为 N , 则其圈复杂度为:

$$V(G) = \sum_{i=1}^N [a(n_i) - 1] + 1 \quad (1)$$

其中

$$a(n_i) = \begin{cases} \text{out deg ree}(n_i), & \text{若 } \text{out deg ree}(n_i) \geq 2 \\ 1, & \text{若 } \text{out deg ree}(n_i) < 2 \end{cases}$$

与公式 $V(G) = P + 1$ 相比, 此公式具有更大的适用性, 能处理含有出度 ≥ 3 的节点的控制流图.

3.2 不可达路径的判定

由于程序中的节点间可能会存在依赖关系, 致使子路径间相互冲突, 因此, 需要对路径集中的路径进行分析, 确定并删除其中的不可达路径. 结合 Soot 工具对源代码处理得到的信息的特点, 可以得到确定不可达路径的基本思想: 通过数据流等分析将程序中的依赖关系转换为基本块间的依赖关系, 将其运用到路径集中, 确定并删除不可达路径. 本步算法如算法 2 所示.

算法 2: 路径集中不可达路径的判定算法

```

输入: 路径集 bPaths
输出: 删除不可达路径后的路径集 bPaths
Begin
1: for (each basispath ∈ bPaths) do
2:   if (basispath 含有冲突子路径链表中的子路径) then
3:     bPaths.remove(basispath);
4:   else
5:     Boolean flag = false; //flag 为循环体判定标记
6:     Block nextNode; //nextNode 为跳出循环体后的第一个节点
7:     for (each node ∈ basispath) do
8:       if (flag) then
9:         if (! node.equals(nextNode)) then
10:           break;
11:         else flag = false;

```

```

12:   endif
13:   endif
14:   for (each unit ∈ node) do
15:     if (unit 为循环语句) then
16:       flag = true;
17:       nextNode = path 中第二个 node 的下一节点;
18:       break;
19:     endif
20:     if (unit 为定义语句) then
21:       if (unit 中定义的变量已经存放于变量表 a 中) then
22:         将新的赋值代替原有的赋值;
23:       else 将 unit 中的变量及其赋值存储到表 a 中;
24:     endif
25:     else if (unit 为判定语句) then
26:       根据 basispath 的下一步分支取向来确定 unit 中各变量之间的关系范围, 并将其存储到表 b 中;
27:       将该取值范围与 a 中相关的变量及其赋值比较;
28:       将该取值范围与 b 中相关的取值范围分析比较;
29:       if (存在冲突) then
30:         将产生冲突的子路径存到冲突子路径链表中;
31:         bPaths.remove(basispath);
32:       endif
33:     endif
34:   endfor
35: endfor
End

```

算法 2 逐一对路径集中的路径分析, 判断当前路径是否含有冲突子路径链表中的子路径(行 1-2): 若含有, 则当前路径为不可达路径, 将其从路径集中除去(行 3); 若不含有, 对该路径的各个节点依次分析, 逐一取节点中的每条语句, 判断该语句是何种类型: 若为循环语句, 则停止对当前节点的分析, 找到跳出该循环的节点, 忽略循环中循环变量的前后数值冲突及循环体内分支语句处的数值依赖与控制依赖关系, 仅对循环体外的节点分析(行 5-17); 若为定义语句, 则将语句中的变量及其赋值存储到表 a 中, 对于同一变量, 若重新赋值, 则将表 a 中的值更新(行 18-22); 若为判定语句, 则根据当前路径的下一步的分支取向来确定该语句中各变量之间的关系范围, 并将其存储到表 b 中, 然后将该取值范围与 a 中相关的变量及其赋值比较, 即进行数值依赖分析, 同时将该取值范围与 b 中相关的取值范围进行分析和比较, 即进行控制依赖分析, 若各存储值间存在冲突, 则说明当前路径不可达, 将其从路径集中除去, 并将路径中的冲突子路径存储到冲突子路

径链表中(行 23—32). 直到路径集中的路径全部判定,算法结束.

算法 2 中包含了数值依赖分析、控制依赖分析以及对循环的处理,下面分别通过例子说明.

(1) 数值依赖分析:如图 1(a)所示的程序段 1,将语句 1、2、3 中的变量及其赋值存储,对于语句 5,在 Soot 中会在该类判定表达式前增加一条语句: $r = a + b$,这样复杂谓词表达式就转换为谓词元表达式 $c < r$. 对于路径 {1, 2, 3, 4, 5, 7}, 由于该路径取节点 5 的假分支,意味着 $c \geq r$,可获取表达式右边 r 的值,得到 $c \geq 3$,这与已存储的 $c = 2$ 冲突,所以该路径为不可达路径.

1: int a=1;	1: ...
2: int b=2;	2: if(a==b)
3: int c=2;	3: ...
4: Boolean flag;	else
5: if(c<a+b)	4: ...
6: flag=true;	5: if(a!=b)
else	6: ...
7: flag=false;	else
...	7: ...
(a) 程序段 1	(b) 程序段 2

图 1 程度段举例

(2) 控制依赖分析:如图 1(b)所示程序段 2,判定语句 2 与 6 之间存在控制依赖关系,对于路径 {1, 2, 4, 5, 7}, 4 为 2 的假分支,该走向意味着 $a \neq b$,将该关系范围存储,而 7 为 5 的假分支,该走向意味着 $a = b$, $a = b$ 与已存储的 $a \neq b$ 产生冲突,所以该路径为不可达路径.

(3) 循环的处理:如图 2 所示的程序段中,循环一次后 i 的值为 1,而执行语句 5 则表示 $i > 5$, i 前后的值冲突,所以循环 1 次的路径为不可达路径,而在寻找基本路径集时不可能按程序中的要求循环 5 次或是更多次,而又必须覆盖循环体内的所有语句,所以对这类情况,在识别不可达路径阶段应特殊化处理.

```

1: int i=0;
2: while(i<=5)
{
3: ...
4: i++;
}
5: System.out.println("结束")

```

图 2 示例程序段

3.3 未覆盖边所在的可达独立路径搜索算法

删除不可达路径后,路径集会存在一些未覆盖

的边,需向路径集中添加这些未覆盖边所在的可达路径. 遍历每一条未覆盖边,搜寻经过该边的路径,为避免搜索到的路径重复,搜索时要避开其它未分析过的未覆盖边(行 1—6). 路径搜索完成后,判断路径是否可达;若可达,则将其添加到路径集中,然后继续搜索经过该边的其它路径(行 7—13);若不可达,继续搜索经过该边的其它路径(行 12). 若经过某未覆盖边的所有路径均不可达,则将搜索到的最后一条不可达路径添加到不可达路径集 1 中(行 14—16). 当所有的未覆盖边已分析完时,停止搜索,算法结束. 本步算法如算法 3 所示.

算法 3: 未覆盖边所在的可达独立路径的搜索算法

```

输入: 路径集 bPaths
输出: bPaths 未覆盖边所在的可达独立路径集 AddFeasiblePath
Begin
1: 找到各判定节点未取的后继节点序列,将其存放到链表数组
   branchSucc 中,数组的长度为控制流图的节点数,判定节点在控
   制流图的块序号即为该节点的后继节点序列在链表数组中的
   行下标;
2: for (each succListi ∈ branchSucc) do
3:   if (succListi.size() > 0) then
4:     Boolean falg=false; //未覆盖边是否找到所在可达独立路
       径的判定标记
5:     for (each sn ∈ succListi) do
6:       搜寻经过 CFG.get(i)→sn 的路径,存储到链表数组
       Paths 中,为避免搜索到得路径重复,搜索时要避开其它
       未分析的覆盖边
7:       for (each path ∈ Paths) do
8:         if (isFeasible(path)) then
9:           bPaths.add(path); AddFeasiblePath.add(path);
10:          flag=true; //当前节点处于循环体内
11:        endif
12:       继续搜索经过 CFG.get(i)→sn 的其它路径;
13:     endfor
14:     if (! flag) then //当前未覆盖边未找到所在可达路径
15:       AddInfeasiblePath1.add(Paths.get(Paths.size()-
16:         1));
17:     endif
18:   endfor
19: endfor
End

```

3.4 不可达独立路径的搜索算法

上一节对路径集未覆盖边所在的可达路径进行了搜索和添加,但对于很多程序,因为不可达路径的存在,程序的可达独立路径数小于圈复杂度,所以对于这类情况,还需继续寻找目前路径集中未覆盖边所在的不可达路径. 此外,为符合第 2 节提到的基本

路径集的性质 3, 还需向路径集中添加为实现线性运算而应覆盖的不可达路径. 所以, 此步中添加的不可达路径有如下两种:

(1) 在完成第三步可达路径的添加后, 路径集 $bPaths$ 仍未覆盖的边所在的不可达路径;

(2) 为实现线性运算而应覆盖的不可达路径, 这类不可达路径主要针对多人入度多出度的节点.

不可达独立路径的具体搜索算法:

(1) 未覆盖边所在的不可达路径的搜索算法

对于该类型不可达路径的搜索, 可在 3.3 节搜索的基础上进行. 在得到未覆盖边所在的可达路径集的同时, 也得到了未找到所在可达路径的未覆盖边所在的不可达路径集 1. 此步只需对未覆盖边遍历(行 1—3), 若路径集 1 中的路径与可达路径集中的某条路径包含同一条未覆盖边, 则将该路径从路径集 1 中删除(行 4—13); 最终得到的路径集 1 即为所求的路径集, 将其中的路径添加到 $bPaths$ 中(行 16—18). 本步算法如算法 4 所示.

算法 4: 未覆盖边所在的不可达路径的搜索算法

```

输入: 路径集  $bPaths$ 
输出:  $bPaths$  未覆盖边所在的不可达独立路径集 AddInfeasiblePath1
Begin
1: for(each succListi ∈ branchSucc) do
2:   if(succListi.size() > 0) then
3:     for(each  $s_n$  ∈ succListi) do
4:       for(each inpath ∈ AddInfeasiblePath1) do
5:         if (AddInfeasiblePath1. contain (CFG. get (i) →  $s_n$ )) then
6:           for(each feapath ∈ AddFeasiblePath) do
7:             if (AddFeasiblePath1. contain (CFG. get (i) →  $s_n$ )) then
8:               AddInfeasiblePath1. remove(inpath); break;
9:             endif
10:          endfor
11:        endif
12:      endfor
13:    endfor
14:  endif
15: endfor
16: for(each p ∈ AddInfeasiblePath1) do
17:    $bPaths$ . add(p); //将路径集 1 中的路径添加到  $bPaths$  中
18: endfor
End

```

(2) 线性运算所需的不可达路径的搜索算法

遍历每一个多人入度多出度节点 m_i (行 1—2), 分析 m_i 的每一个前驱, 查看是否存在前驱节点 $pred_j$,

使得路径集 $bPaths$ 中包含边 $pred_j \rightarrow m_i$ 的各路径构成的子路径集能完全覆盖 m_i 的所有后继节点(行 3—15); 若存在, 则说明对于 m_i , 不需要再添加路径(行 22); 若不存在这样的前驱节点, 则将未覆盖的后继个数最少的前驱节点及未覆盖后继节点序列存储起来(行 16—21), 然后设计搜索算法, 对已存储的每一个未覆盖后继节点搜索得到一条连续通过已存储的前驱节点、 m_i 及该后继节点的路径, 并将其加到不可达路径集 2 中(行 25—27). 当所有的多人入度多出度节点均分析完毕, 将不可达路径集 2 中的所有路径链表添加到 $bPaths$ 中(行 29—31), 算法结束. 本步算法如算法 5 所示.

算法 5: 线性运算所需的不可达路径的搜索算法

```

输入: 路径集  $bPaths$ 
输出:  $bPaths$  线性运算所需添加的不可达独立路径集 AddInfeasiblePath2
Begin
1: 找到 CFG 中所有的多人入度多出度节点, 存放到链表 MultiNode 中;
2: for(each  $m_i$  ∈ MultiNode) do
3:   List<Block> Pred = CFG. getPredsOf( $m_i$ );
4:   List<Block> Succ = CFG. getSuccsOf( $m_i$ );
5:   for(each  $pred_j$  ∈ Pred) do
6:     for(each path ∈  $bPaths$ ) do
7:       if( $pred_j, m_i$  相邻存在于 path 中) then
8:         for(each succ ∈ Succ) do
9:           if(succ ∈ path) then
10:            Succ. remove(succ);
11:          endif
12:        endfor
13:      endif
14:    endfor
15:    if(Succ.size() != 0) then
16:      if( $j=0$ ) then
17:        将  $pred_j$  与 Succ 中的节点添加到 UnSucc 中;
18:      else if(Succ.size() < UnSucc.size() - 1) then
19:        UnSucc 清空并重新赋值;
20:      endif
21:    endif
22:    else break;
23:  endif
24: endfor
25: for(each  $s$  ∈ UnSucc. remove(0)) do
26:   找到一条连续经过 UnSucc. get(0)、 $m_i$  和  $s$  的路径, 并将其添加到 AddInfeasiblePath2;
27: endfor
28: endfor
29: for(each p ∈ AddInfeasiblePath2) do
30:    $bPaths$ . add(p); //将路径集 2 中的路径添加到  $bPaths$  中
31: endfor
End

```

以三角形分类程序为例来验证本文提出的方法的有效性。

4 实验结果分析

实验部分除用三角形分类程序来验证文中提出的方法外,还对其它程序进行了分析,得到的实验数据如表 1 所示。

经验证,表中得到的结果与人工分析得到的结

果一致。在不可达路径识别上,较以往的静态分支相关检测方法,本文提出的方法所能检测的分支情况占程序中分支总数的百分比有了较大提高。此外,通过该方法最终得到的基本路径集中的可达路径数有了明显提升。实验结果表明,本文提出的基本路径集生成方法能有效地生成包含尽可能多的可达路径的基本路径集。

表 1 基本路径集生成方法实验结果

程序名称	节点数	分支节点数	检测的分支数占 分支总数的百分比	基本路径数	可达路径数	不可达路径数
三角形分类	16	9	100%	10	8	2
Fibonacci 数列	8	3	100%	4	4	0
冒泡排序	12	4	100%	5	5	0
杨辉三角问题	16	5	100%	6	6	0

5 结束语

本文提出的基本路径集生成方法,在保证生成的路径集为基本路径集的前提下,能使路径集包含尽可能多的可达路径,在一定程度上提高了软件测试的效率。

参考文献:

[1] B.Beiter. Software system testing[M]. Van Nostrand Reinhold Company, 1990.

[2] James R Birt, Renate Sitte. Optimizing testing efficiency with error prone path identification and genetic algorithms[C]//2004 Australian Software Engineering Conference (ASWEC'04), Melbourne, Australia: IEEE, 2004:106-115.

[3] Antonia Bertolino, Martina Marre. Automatic generation of path covers based on the control flow analysis of computer programs[J]. IEEE Transaction on software Engineering, 1994,20(12):885-899.

[4] 王冠,景小宁,王彦军. 基本路径测试中的 McCabe 算法改进与应用[J]. 哈尔滨理工大学学报,2010,15(1): 48-51.

[5] DU Qing-feng, DONG Xiao. An improved algorithm for basis path testing[C]//The International Conference on Business Management and Electronic Information (BMEI2011), Guangzhou, China: IEEE, 2011: 175-178.

[6] 刘勇,曾明,朱利,等. 基于数据流的软件测试序列自动

生成技术研究[J]. 微电子学与计算机,2005,22(5): 131-135.

[7] Roger Pressman. Software engineering: a practioner's approach[M]. Fifth Edition, Boston: McGraw Hill, 2001:59-61.

[8] 李必信. 程序切片技术及其应用[M]. 北京:科学出版社,2006.

[9] Vallee-Rai R. Soot: A Java Bytecode Optimization Framework10/2000[EB/OL]. [2012 - 05 - 29]. <http://www.sable.mcgill.ca/publications/thesis/Reference/>.

作者简介:



韩 寒 女,(1989—),硕士研究生. 研究方向为软件测试、路径分析。



姜淑媚 女,(1966—),博士,教授,博士生导师,CCF 会员. 研究方向为软件工程、软件测试、程序设计语言。