从 GCC 的 AST 文本提取 C 源程序静态信息的方法

封战胜，苏小红，马培军

（哈尔滨工业大学 计算机科学与技术学院，哈尔滨 150001，fengzhansheng1984@126.com）

摘 要：为了能够正确的分析源程序的控制依赖关系和数据依赖关系，以便在此基础上进行程序切片及冗余代码和重复代码检测，提出一种利用 GCC 抽象语法树（AST）文本来提取源程序静态信息的方法. 首先，对 GCC AST 文本进行标准化及消除文本中与控制流分析和数据流分析无关的结点信息；其次，构建控制依赖子图；同时如果需要数据流分析，在控制依赖子图的基础上构建控制流图，在控制流图的基础上构建数据流子图；最后通过引入过程间分析来完善系统依赖图. 实验结果表明，这种方法基本能正确的分析源程序的控制依赖和数据依赖关系，具有更好的适应性和灵活性.

关键词：程序静态分析；AST；控制依赖；数据依赖；控制流图；系统依赖图

中图分类号：TP311 **文献标志码**：A **文章编号**：0367－6234(2010)07－1100－04

Extraction of static information of C program from GCC abstract syntax tree text

FENG Zhan-sheng, SU Xiao-hong, MA Pei-jun

(School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China, fengzhansheng1984@126.com)

Abstract: In order to correctly analyze the control dependence and data dependence relations of C program, and then carry out the program slicing, code redundancy and code duplication detection, a method to extract the static information of C program from GCC AST text was put forward. Firstly, the GCC AST text was standardized and the nodes unrelated to control dependence and data dependence analysis were eliminated. Secondly, control dependence subgraph was constructed. If data flow analysis was necessary, control flow graph was constructed based on control dependence subgraph and data dependence subgraph was constructed based on control flow graph. At last, by the introduction of inter-process analysis, system dependence graph was improved. It is indicated that the research can correctly analyze the control dependence and data dependence relations.

Key words: program static analysis; abstract syntax tree (AST); control dependence; data dependence; control flow graph; system dependence graph

静态信息描述软件在源代码中的结构，是对软件组件及组件之间关系的信息描述，可以帮助开发人员和维护人员进行程序理解. GCC 的抽象语法树是源程序的一种中间表示形式，比较直观的表示出源程序的语法结构，并含有源程序结构显示所需要的全部静态信息. 解析方法一般分为两种：1）通过编译的方法构造分析器对抽象语法树文本进行解析[1－3]；2）基于 XML 或者 GXL 的解析方法[4－5]. 以往的研究大多没有消除抽象语法树文本中的冗余信息，构建系统依赖图时在控制流图的基础上分别进行控制依赖和数据依赖分析. 本文提出了抽象语法树标准化及消除冗余结点的算法，以降低文本解析的复杂程度. 同时提出了构建系统依赖图的一种新流程，先构建控制

收稿日期：2009－07－01.
基金项目：国家自然科学基金资助项目(60673035).
作者简介：封战胜(1984—)，男，硕士研究生；
苏小红(1966—)，女，教授，博士生导师.

依赖子图，如果需要数据流分析，在控制依赖子图的基础上构建控制流图，在控制流图的基础上构建数据流图，从而使该方法具有更好的适应性和灵活性.

1　静态信息提取模型

基于 GCC 抽象语法树文本的静态信息提取模型如图 1 所示.

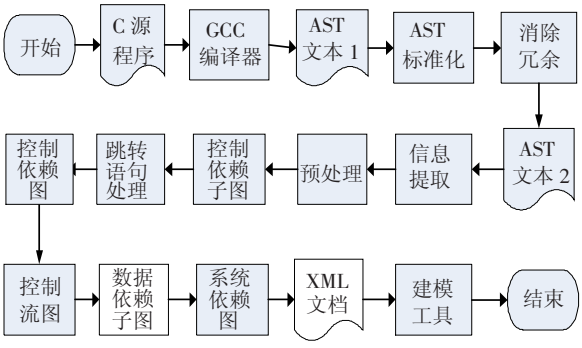

图 1　静态信息提取流程

GCC 编译器生成的 AST 文本中结点表示格式不规范，文本中包含许多有助于编译的细节信息，但这些信息与控制依赖分析和数据依赖分析无关.

1）对 GCC 抽象语法树文本进行标准化和消除文本中与控制依赖分析和数据依赖分析无关的冗余信息，然后结合面向对象思想，进行结点静态信息提取.

2）把控制依赖分析归结为求取语句结点的父亲－孩子信息，构建控制依赖子图；

3）如果用户需要数据依赖分析，在控制依赖子图的基础上构建控制流图，把构建控制流图归结为求取结点的前驱和后继信息，在控制流图的基础上构建数据依赖子图，把数据依赖分析归结为求取结点的到达－定值信息. 通过引入过程间分析，来完善系统依赖图.

4）通过输出系统依赖图的结点信息和结点之间的关系，调用建模工具，以图形的形式展示给用户.

2　GCC AST 文本标准化及消除文本中的冗余信息

标准化 AST 文本就是把一个 AST 结点统一用一行字符串表示，把由两个单词构成的标记标签合并成一个单词，如果标记值为空则把标记标签统一用"attr"表示.

为了消除 AST 文本中的冗余信息，首先把 AST 结点归类为有用结点、无用结点、不确定结点，即种子结点. 有用结点[6] 即结点含有 srcp 标记标签，且其标记值在源程序文件名集合中. 无用结点即结点含有 srcp 标记标签，且其标记值不在源程序文件名集合中. 不确定结点即结点不含有 srcp 标记标签. 然后，利用种子结点来传播，把不确定结点集合中的元素再分类，根据有用结点的分支结点是有用结点，无用结点的分支结点为无用结点，如果不确定结点的父亲结点集合中有有用结点，则把其归类为有用结点，直到不确定结点集合不再发生变化为止. 最后只需要根据结点标记符 call_expr 找回源程序调用的系统函数.

3　结点信息提取及预处理

根据 GCC 抽象语法树结点所代表的对象类型的不同，主要分为 5 类：常量结点（后缀为 cst）、类型结点（后缀为 type）、声明结点（后缀为 decl）、语句结点（后缀为 stmt）、表达式结点（后缀为 expr）.

C 语言语句主要分为简单语句和复合语句，其中简单语句包含跳转语句和非跳转语句.

定义 1　把 decl_stmt、expr_stmt、break_stmt、continue_stmt、goto_stmt、scope_stmt、label_decl 归类为简单语句，其中，break_stmt、continue_stmt、goto_stmt 归类为跳转语句.

定义 2　把 if_stmt（if 语句的肯定分支）、else_stmt（if 语句的否定分支）、switch_stmt、case_stmt、for_stmt、while_stmt、do_stmt 归类为复合语句.

控制依赖子图的生成要用到语句的范围，其中简单语句的开始范围和结束范围是相同的. AST 文本给出了语句（除 case_stmt）的开始范围，利用语句结点的标记标签 next 求取复合语句的结束范围，然后利用已求得的语句范围来确定 case_stmt 的范围. 为了取消循环语句的多样性，可以把循环语句统一表示成一种表现形式. 函数调用结点中的实参可能是表达式或者存在函数嵌套，可以把其标准化成若干个 CallNode 结点，其中，Callnode 包含实参列表、返回值和指向调用的函数的入口等信息.

4　控制依赖分析和控制依赖子图的生成

控制依赖主要是由分支语句和循环语句等引起的.

定义 3　设 u 和 v 为程序中的两条语句，若在 u 执行时对其控制条件的不同取值直接决定着 v 是否执行，则 u 和 v 之间有一条控制依赖边，即 u

为 v 的父亲结点,v 为 u 的孩子结点, 用 father-child(u,v) 表示.

生成控制依赖子图的具体算法描述如图 2 所示.

```
算法: 生成控制依赖子图 CDSG
输入: 函数语句列表 TheList    输出: CDSG
begin
    TheStack..push(TheList.front());
    while(!TheStack.empty())
        TopStmt=TheStack.top()//[ F1, F2];
        TheStmt=TheList.next();//[ S1, S2]
        if(!scope_stmt)
            if([ S1, S2] in [ F1, F2])
                father-child(TopStmt,TheStmt);
                if(TheStmt 为复合语句)
                    TheStack.push(TheStmt);
            else
                TheStack.pop();
                while(!TheStack.empty())
                    TopStmt=TheStack.top()//[ F1, F2];
                    if([ S1, S2] in [ F1, F2])
                        father-child(TopStmt,TheStmt);
                        if(TheStmt 为复合语句)
                            TheStack.push(TheStmt)
                            break;
                        else
                            TheStack.pop();
                        endif
                    endwhile
                endif
        else
            if(scope_stmt.attribute==end)
                TheStack.pop();
        endif
    endwhile
end
```

图 2 生成控制依赖子图 CDSG 的算法

5 控制流图的生成

控制流图的生成主要是求出每条语句的前驱集合和后继集合,生成控制流图的算法如图 3 所示.

```
算法: 生成控制流图 CFG
输入: 控制依赖子图 CDSG   输出: 控制流图 CFG
begin
    foreach CDSG's statement TheStmt
        switch(TheStmt->kind)
            case expr_stmt/decl_stmt/CallNode
                if(!复合语句's furthest right child)
                    Pre-Suc(TheStmt, NextStmt);
            endcase
            case break_stmt/continue_stmt/goto_stmt
                Pre-Suc(TheStmt, JumpStmt);
            endcase
            case for_stmt/while_stmt/do_stmt/if_stmt/case_stmt
                Pre-Suc(TheStmt, FirstChild),并标记为 T;
                Pre-Suc(TheStmt, LeapStmt),并标记为 F;
                Process the furthest right child;
            endcase
            case switch_stmt
                Pre-Suc(TheStmt,FirstChild);
            endcase
            case  else_stmt
                Pre-Suc(TheStmt,FirstChild);
                Process the furthest right child;
            endcase
        endswitch
    endfor
end
```

图 3 生成控制流图 CFG 的算法

定义 4 控制流图是一个有向图 $G=(V,E)$,其中,V 为结点的集合,结点对应源程序的语句,E 为有向边,令$(x,y)\in E$ 表示控制流图中的边 $x\rightarrow y$, 称 x 是 y 的前驱,y 为 x 的后继, 用 Pre-Suc(x,y) 表示.

6 数据依赖分析和数据依赖图的生成

数据流依赖边一般表示变量之间的定义 - 引用关系,它包括直接数据流依赖和间接数据流依赖. 数据流信息可以通过建立和解方程来收集,这些方程联系程序中不同点的信息,数据流方程[7]如表 1 所示. 本文对传统的数据流迭代算法进行了改进,对于大型程序,首先把程序分成基本块.

定义 5 基本块是一个连续的语句序列,它只有一个入口和一个出口,连续的不在循环语句内或者分支语句内的简单语句合并为一个基本块,两个基本块之间的语句划分为一个基本块.

本文对每个过程 P 定义集合 CHANGE [P], 它的值是在执行 P 期间可能改变的全局变量和 P 的形式参数的集合. 过程间数据流方程定义如表 2 所示,具体算法描述如图 4 所示.

表 1 数据流方程的定义

变量	定义
GEN[N]	GEN[N] = { < x,N > | n 定值了 x}
KILL[N]	KILL[N] = { < x,M > | M 和 N 分别定值了 x, $M\neq N$ 且 M 可到达 N}
IN[N]	IN[N] = ∪OUT[P] // P 是 N 的前驱结点
OUT [N]	OUT [N] = GEN [N] ∪(IN [N] - KILL [N])

表 2 过程间数据流方程的定义

变量	定义
DEF [P]	DEF [P] = { < x,P > | x 为 P 的形参或者全局变量,P 定值了 x}
PARM [P - Q]	PARM [P - Q] = {x | x 是可能改变的变量, x 作为 P 调用 Q 的第 i 个实参,且 Q 的第 i 个形参在 CHANGE[Q]中}
GLOB [Q]	GLOB [Q] = {x | P 调用 Q,x 在 CHANGE[Q] 中是全局的}
CHANGE [P]	CHANGE [P] = DEF [P] ∪PARM [P - Q] ∪ GLOB [Q]

```
算法: 计算每个过程 P 的 CHANGE[P]
输入: 过程 P1, P2, ...Pn    输出: 对每个过程P, 生成 CHANGE[P]
begin
    for(每个过程 P)
        CHANGE[P]=DEF[P]
    endfor
    change=true;
    while(change)
        change=false;
        for(i=0;i<n;i++)
            OldChange[Pi]=CHANGE[Pi];
            for(Pi 调用每个过程 Q)
                把 CHANGE[Q]的全局变量加到 CHANGE[P]中;
                for(Q 的每个形式参数 x(自左向右数第 j 个))
                    if (x 属于 CHANGE[Q])
                        for(Pi 的每个过程调用 M)
                            if(M 的第 j 个实参 x 是可能改变的变量且 x 是Pi 的形参)
                                把 x 加到 CHANGE[Pi]中;
                            endif
                        endfor
                    endif
                endfor
            endfor
            if(OldChange[Pi]!=CHANGE[Pi])
                change=true
            endif
        endfor
    endwhile
end
```

图 4 计算每个过程 P 的 CHANGE [P] 的算法

7 实验结果分析

图 5 列举了一个时钟显示的源程序,图 6 是生成的系统依赖图,表 3 中的源程序[8]涵盖了较

多的 C 语言语法，图 6 的结果表明该方法基本能够正确分析源程序的控制依赖和数据依赖关系，从表 3 中可以看出该方法能正确的分析源程序的控制依赖关系，数据依赖分析存在误差，主要是因为变量别名分析和指针分析模块不够完善，从而影响了数据依赖分析的精度[9-12].

```
1  struct  CLOCK
2  {
3     int hour;
4     int minute;
5     int second;
6  }
7  void Update(CLOCK* t)
8  {
9     t->second++;
10    if(t->second==60)
11    {
12       t->second=0;
13       t->minute++;
14    }
15    if(t->minute==60)
16    {
17       t->minute=0;
18       t->hour++;
19    }
20    if(t->hour==24)
21    {
22       t->hour=0;
23    }
24 }
25 void Display(CLOCK* T)
26 {
27 printf(t->hour,t->minute,t->second)
28  }
29  void Delay()
30  {
31     long i=0;
32     while(i<50000000)
33     {
34        i++;
35     }
36  }
37  int main()
38  {
39     long j=0;
40     CLOCK m;
41     m hour=0;
42     m minute=0
43     m second=0;
44     while(i<1000000)
45     {
46        Update(&m);
47        Display(&m);
48        Delay();
49        i++
50     }
51     return 0;
52  }
```

图 5　时钟显示源程序

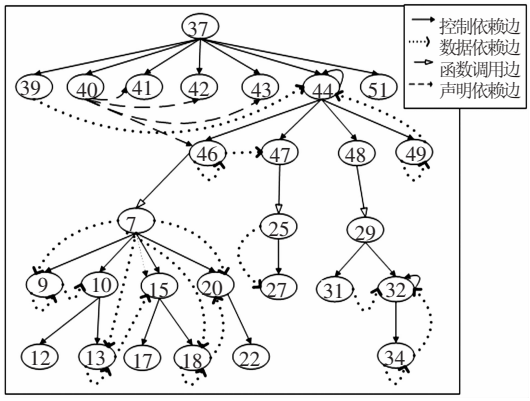

图 6　时钟显示源程序对应的系统依赖图

表 3　源程序的控制依赖分析和数据依赖分析正确率　%

程序	控制流	数据流
1	100	100.0
2	100	100.0
3	100	96.7
4	100	46.7
5	100	86.7
6	100	94.3
7	100	100
8	100	43.6
9	100	87.7
10	100	97.4

8　结　论

1）针对目前 GCC 抽象语法树文本解析复杂程度高的问题，提出了抽象语法树文本标准化及消除冗余信息的算法，降低了文本分析的复杂程度；

2）同时提出了构建系统依赖图更加合理的方法，具有更强的适应性和灵活性.

参考文献：

[1] ANTONIOL G, Di PENTA M, MASONE G, *et al.* Compiler hacking for source code analysis[C]//Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation. Washington, DC: IEEE Computer Society, 2004:383 - 406.

[2] 李慧贤. 面向对象程序切片中的控制流分析[D]. 西安:西安电子科技大学,2003.

[3] 陆仲达. 面向对象程序切片中的数据流分析[D]. 西安:西安电子科技大学,2003.

[4] ANTONIOL G, Di PENTA M, MASONE G, *et al.* XML-oriented gcc ast analysis and transformations [C]//Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation. Washington: IEEE Computer Society, 2003:173 - 182.

[5] HOLT R C, SCHURR A, SIM S E, *et al.* GXL: A graph-based stardard exchange format for reengineering [J]. Science of Computer Programming, 2006,60(2): 149 - 170.

[6] 李鑫,王甜甜,苏小红,等. 消除 GCC 文本抽象语法树中冗余信息的算法研究[J]. 计算机科学, 2008, 35(10):170 - 172.

[7] DEARMAN D, COX A, FISHER M. Adding control-flow to a visual data-flow representation[C]//Proceedings of the 13th International Workshop on Program Comprehension. Washington: IEEE Computer Society, 2005: 297 - 306.

[8] 苏小红, 孙志岗. C 语言大学实用教程学习指导[M]. 北京:电子工业出版社, 2008.

[9] STOREY M A. Theories, methods and tools in program comprehension: Past present and future[C]//Proceedings of the 13th International Workshop on Program Comprehension. Washington: IEEE Computer Society, 2005: 181 - 191.

[10] KOTHARI S C. Scalable program comprehension for analyzing complex defects[C]//Proceedings of the 16th IEEE International Conference on Program Comprehension. Washington: IEEE Computer Society, 2008: 3 - 4.

[11] MALETIC J I, KAGDI H. Expressiveness and effectiveness of program comprehension: Thoughts on future research directions[C]//Proceedings of the 16th Frontiers of Software Maintenance. Beijing: FoSM, 2008: 31 - 37.

[12] ZHI L L, WANG J, ZENG Q K. Key technologies and improvement of identifying covert channels in source code [J]. Journal of Shanghai Jiaotong University, 2009,43(1):101 - 105.

（编辑　张　红）