面向程序理解的系统依赖图构建算法

王克朝[1,2]，王甜甜[1]，苏小红[1]，马培军[1]，童志祥[1]

(1. 哈尔滨工业大学 计算机科学与技术学院，150001 哈尔滨；2. 哈尔滨学院 软件学院，150086 哈尔滨)

摘 要： 为降低程序理解中的程序标准化和程序匹配等复杂度，提出了面向程序理解的系统依赖图构建算法，将其划分为3个阶段：程序信息的提取、控制依赖子图的构建和数据依赖子图的构建. 采取控制依赖和数据依赖分别求解，直接基于控制依赖子图分析数据流，无需额外的控制流图表示，并且可按需计算数据流，降低了算法复杂度；将选择语句和循环语句统一表示，并将表达式表示为抽象语法树，使之便于程序转换和分析. 在编程题自动评分系统和程序识别中的应用中结果表明构建的系统依赖图为程序理解和分析提供了方便，降低了复杂度.

关键词： 程序理解；系统依赖图；程序信息提取；控制依赖；数据依赖

中图分类号： TP311 **文献标志码：** A **文章编号：** 0367-6234(2013)01-0078-07

Program comprehension oriented construction algorithm of system dependence graph

WANG Kechao[1,2], WANG Tiantian[1], SU Xiaohong[1], MA Peijun[1], TONG Zhixiang[1]

(1. School of Computer Science and Technology, Harbin Institute of Technology, 150001 Harbin, China;
2. School of Software, Harbin University, 150086 Harbin, China)

Abstract: To reduce the complexity of program comprehension such as program standardization and program matching, a program comprehension oriented construction algorithm of system dependence graph is proposed. This algorithm can be divided into three stages: program information extraction, control dependence sub-graph construction and data dependence sub-graph construction. Control dependency and data dependency are independently computed. Data flow is directly analyzed based on control dependent sub-graph without extra control flow graph, and computed on demand, which reduces the algorithm complexity. Selection statements and loop statements are uniformly represented, and expressions are represented as abstract syntax trees, which can facilitate program transformation and analysis. The system dependence graph has been used in automatic grading system of student programs and program recognition. Application results show that it can facilitate program comprehension and analysis and reduce the complexity of program comprehension and analysis.

Key words: program comprehension; system dependence graph; program information extraction; control dependence; data dependence

程序理解[1]是通过对程序的分析、抽象及推理获取知识的一个演绎过程，需要解决数据收集、知识组织及信息浏览3个方面的问题. 解决这些问题的一个关键是程序的中间表示.

K. J. Ottenstein 等[2]提出了程序依赖图的概念. S. Horwitz 等[3]将程序依赖图扩展为系统依赖图，捕获过程间的调用关系. 系统依赖图是程序基于图形的中间表示，能清晰地表示出程序中的各种控制依赖和数据依赖关系及程序的语法与语义信息，因此在编译器优化[4-5]、程序切片[6]、软件

收稿日期：2012-02-03.
基金项目：国家自然科学基金资助项目(61202092,61173021)；高等学校博士学科点专项科研基金资助项目(20112302120052)；中央高校基本科研业务费专项资金资助项目(HIT.NSRIF.201178)；黑龙江省高教学会“十二五”重点规划课题资助项目(HGJXH B1110957).
作者简介：王克朝(1980—)，男，博士研究生；
苏小红(1966—)，女，教授，博士生导师；
马培军(1963—)，男，教授，博士生导师.
通信作者：王甜甜，sweetwtt@126.com.

审查[7]、软件测试[8]、编程题自动评分[9-11]、克隆代码检测[12-13]以及软件缺陷检测[14-15]等领域得到了广泛的应用. 系统依赖图的构建方法也得到了广泛的研究，近年来提出了面向对象[16]、面向方面[17]以及函数式程序[18]的系统依赖图的创建方法.

然而，程序理解过程中，常常需要按需的程序分析，如果程序规模较大，则很难一次理解、分析所有的系统依赖图节点及边; 此外，由于编程语言的语法多样性，同一个编程任务可由多种语法结构实现，这种现象称为代码多样化，增加了程序理解的难度. 因此，本文采取控制依赖和数据依赖分别求解，在控制依赖子图的基础上，按需进行数据流分析，降低了算法复杂度，使之适合于分析大规模代码; 将选择语句和循环语句统一表示，将表达式表示为抽象语法树，为实现程序标准化时进行语义等价的程序转换和程序匹配提供了方便.

1　系统依赖图的概念

定义 1　程序依赖图(Program Dependence Graph，PDG) 单过程程序 P 的 PDGG = (V,E) 是一个有向图. 其中: V 为节点集合，表示程序中的语句和谓词; $E \subset V \times V$ 为边集合，表示节点间的数据依赖或控制依赖关系.

定义 2　系统依赖图(System Dependence Graph，SDG) 多过程程序 P 的 SDGG = {GPDGs，EInterpro} 是一个有向图. 其中: GPDGs 为 PDG 集合，每个过程(函数) 表示为一个 PDG; EInterpro 为过程间调用边与依赖边的集合，过程间的调用边将函数调用位置与被调函数的 entry 相连，过程间的数据依赖边表示实参和形参间的数据流.

系统依赖图包含控制依赖子图和数据依赖子图.

定义 3　控制依赖子图(Control Dependence Sub-graph，CDS) 由节点和控制依赖边构成，表示程序的控制流信息.

定义 4　数据依赖子图(Data Dependence Sub-graph，DDS) 由节点和数据依赖边构成，表示程序的数据流信息.

依赖图中的边主要有 4 种依赖关系: 控制依赖、流依赖、声明依赖和过程调用依赖. 令 v_1，v_2 分别为依赖图中的两个节点，分别定义如下:

定义 5(控制依赖)　如果 v_2 的执行是由 v_1 表示的谓词在执行时确定的，那么 v_2 控制依赖于 v_1. 不从属于任何控制谓词的节点控制依赖于 entry 节点.

定义 6(流依赖)　如果有一条从 v_1 到 v_2 的路径，且存在一个在 v_2 中定义并在 v_1 中使用的变量，且该变量在从 v_1 到 v_2 的路径上的其他任何地方没有被重新定义，则称 v_2 流依赖于 v_1.

定义 7(声明依赖)　从声明某变量的节点 v_1 到后面的各个定义该变量的节点 v_2 间的特殊的数据依赖.

定义 8(过程调用依赖)　从函数调用节点 v_1 到被调函数入口节点 v_2 的边.

2　面向程序理解的 SDG 自动生成模型

2.1　对传统 SDG 的改进

1) 传统 SDG 数据流分析的复杂度高，不适合于大规模软件分析，因此本文将 SDG 分解为 CDS 和 DDS，基本操作基于 CDS，只在需要数据流信息时，在 CDS 的基础上计算数据流信息，这样既可利用 SDG 能够表示程序语法与语义信息及便于程序匹配的优点，又可降低数据流分析的复杂度，节省空间和时间，使之适合于分析大规模代码.

2) 传统 SDG 以不同的形式表示各种选择结构语句和循环语句，不利于代码标准化操作，因此本文在 SDG 中新增加了选择结构节点和循环结构节点类型. 选择结构节点包括 selection 节点和 selector 节点，统一了所有选择结构语句 if、if-else、switch 和选择运算符(?:) 的表示形式，其中: selection 节点为选择结构; selector 节点及其子节点为选择分支. 循环结构节点包括 iteration 节点和 init_sub 节点，统一了 for、while 和 do-while 的表示形式. 其中 iteration 节点表示选择结构，其子节点表示循环体中的语句，init_sub 节点用于 do-while 语句中. 统一了选择语句和循环语句表示，便于消除选择结构和循环结构的语法表示的多样化.

3) 传统的 SDG 将表达式表示为节点中的 token 串，这种表示方式不能充分表示表达式的语法结构，本文将表达式表示为抽象语法树，连接到语句节点上. 这样，使得控制依赖树不但可以表示控制流，还可以充分表示语法信息，便于执行指针分析和程序转换.

2.2　自动生成模型

根据源程序 SDG 的特点，本文基于程序理解的基本原理将整个 SDG 自动生成过程划分为: 程序信息的提取、CDS 的构造和 DDS 的构造. 图 1 给出了面向程序理解的 SDG 自动生成的模型.

程序信息提取是构造 SDG 的基础，通过词法和语法分析提取输入源程序的有用信息，创建源

程序的 token 串、符号表和函数列表，同时处理程序的一些词法和语法错误.

CDS 是根据程序信息提取阶段创建的 token 串、符号表和函数列表来构建的，在创建系统 CDS 的同时处理程序的某些语法错误.

DDS 的构造本质上是建立流依赖边和声明依赖边，可归结到程序中到达 - 定值信息的求解. 由于 CDS 包含控制流信息，可通过 CDS 获得节点之间的控制依赖关系，从而获得各个节点的到达 - 定值信息，因此可在创建完 CDS 后对其执行数据流分析，构建 DDS.

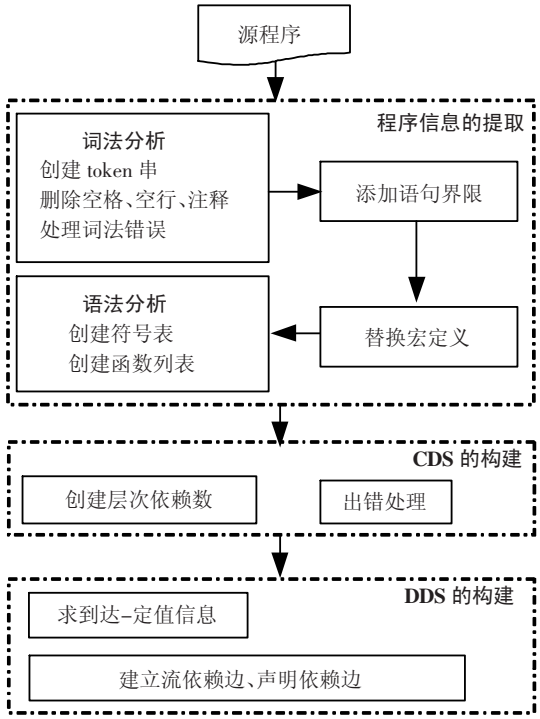

图1 面向程序理解的 SDG 自动生成模型

3 CDS 的构建

进一步分析程序信息提取阶段创建的 token 串、符号表和函数列表来构建 CDS，算法如图 2 所示. 其中: GS 为语法栈，用来保存分析过程中需要记录的语法信息; hQ 为控制依赖节点队列，用以记录节点间的控制依赖关系. 该算法识别程序的语法元素，并建立各个语句节点间的控制依赖关系，从而生成 CDS.

4 按需 DDS 构建

数据依赖是节点表示的语句之间数据的定义和使用关系. 数据依赖可分为不同的类型，如流依赖、声明依赖等.

本文自底向上按需分析给定程序模块的函数调用关系: 对给定程序模块的函数调用图，首先计算其终端节点函数的数据依赖，然后自底向上分析该函数的主调函数的数据依赖，当终端节点存在多个主调函数时，只需分析终端节点一次，无需重复分析终端节点，也无需多个过程依赖图拷贝. 可以对给定程序模块执行局部数据依赖分析，因此适用于分析大规模程序.

基于编译器代码优化领域中经典的迭代数据流分析算法，在 CDS 的基础上计算数据依赖信息，建立 DDS，其构建模型如图 3 所示.

```
算法: CDS 的构建.
输入: 源程序对应的 token 串 iTstr、符号表和函数列表.
输出: 源程序对应的 CDS.
{
  InitStack( GS) ,标记其栈顶指针为 TOP;
  创建一个 head 节点并将其索引入栈, InitQueque( hQ) ;
  while( GS 不空)
  {
    If( ( 包含 ‖ 宏定义 ‖ 变量声明 ‖ 函数声明) 语句的标志)
    创建相应类型的节点并入队 hQ;
    IDX = 该语句结束位置;
  Else If( 函数入口标志)
    创建 entry 节点并入队 hQ;
    entry 节点入栈 GS;
    IDX = 函数开始, TOP 指向该节点;
    if( 此函数为带参函数)
      对每个参数创建 vdeclare 节点并入队 hQ;
  Else If( 其他语句的标志)
    创建相应类型的节点并入队 hQ;
    IDX = 语句结束位置;
    If( 节点在 iTstr 中的位置处于 TOP 所指节点在 iTstr 中的位置范围内)
      节点入队 TOP 所指节点的队列;
    Else
      出栈直到节点在 iTstr 中的位置处于 TOP 所指节点在 iTstr 中的位置范围内;
      节点入队 TOP 所指节点的队列;
    If( 当前是简单语句节点)
      IDX = 该语句的结束位置;
      TOP 指向栈顶元素的结束位置;
    Else if( 当前是复合语句节点)
      IDX 指向此语句体的开始位置;
      节点索引入栈 GS;
      TOP 指向栈顶元素对应的节点;
  }
}
```

图2 系统 CDS 构建算法

4.1 求到达 - 定值信息

要建立各种数据依赖边应首先求节点间的到达 - 定值信息. 到达 - 定值的相关概念如表 1

所示.

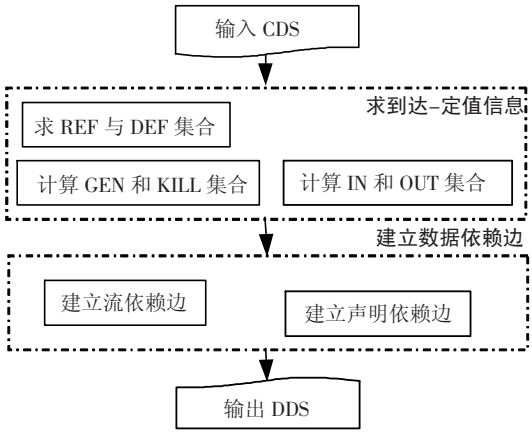

图 3　系统 DDS 构建模型

表 1　数据流方程的定义

名称	定义	说明
数据流方程	OUT [N] = GEN [N] ∪ (IN [N] - KILL [N])	N 对应的语句后一点的到达 - 定值集合
GEN [N]	GEN [N] = { < x,N >: N 定值了 x}	CDS 中节点 N 对应语句产生定值集合
KILL [N]	KILL [N] = { < x,N_1 >: N 定值了 x,N_1 ! = N 且 N_1 是 N 的前驱节点}	N 对应语句在程序中注销的定值集合
IN [N]	IN [N] = ∪OUT (precessor(N)), precessor(N) 是 N 的数据流前驱结点	N 对应的语句前一点的到达 - 定值集合

其中:

1) 变量 x 的定值是一个语句，它赋值或可能赋值给 x. 最普通的定值是对 x 的赋值或读值到 x 中的语句.

2) 如果有路径从紧跟 d 的点到达 p，并且在这条路径上 d 没有被注销，则定值 d 到达 p.

3) 如果某个变量 a 的定值 d 到达点 p，那么 p 引用 a 的最新定值可能在 d 点. 如果沿着这条路径的某两点间是读 a 或对 a 的赋值，那么注销变量 a 的定值 d.

为了计算到达程序中每个语句的定值的集合，首先计算局部的定值信息，然后通过控制依赖边进行传播. 数据流信息可以由建立和解方程来收集，这些方程联系程序不同点的信息.

当控制流通过一个语句时，语句末尾的信息是在这个语句中产生的信息或是进入语句开始点并且没有被这个语句注销的信息. 由数据流方程可看出，算法的关键在于各个控制流节点的 GEN 和 KILL 集合的计算，而生成 GEN 和 KILL 集合则需要知道各个节点定值和使用了那些变量，用 DEF 与 REF 集合来表示.

4.1.1　计算 REF 与 DEF 集合

CDS 中节点的 GEN、KILL、IN、OUT 集合的计算都是基于各节点的 REF 与 DEF 信息的，所以从 CDS 的节点中收集 REF 与 DEF 信息对于到达 - 定值的计算是十分关键的.

CDS 中每个节点都有一个 REF 和 DEF 集合. 一个节点的 REF 集合包含了该节点进行计算时引用的所有变量，DEF 集合包含了该节点计算的变量. 如表 2 所示.

表 2　基本表达式的 REF 集合定义

序号	表达式类别	REF 集合定义
1	常数	REF(const) = Ø
2	变量	REF(var) = { var}
3	一元	REF(unary _ op exp) = REF(exp)
4	二元	REF(exp1 binary _ op exp2) = REF(exp1) ∪ REF(exp2)

在建立好 CDS 后，便可以后根序遍历 CDS，根据节点的不同类型，采用表 3 中相应的表达式求其 REF 和 DEF 集合.

表 3　各种不同类型的语句的 REF 和 DEF 集合定义

语句类别	REF 集合定义	DEF 集合定义
赋值	REF(var = exp) = REF(exp)	DEF(var = exp) = { var}
scanf	REF(scanf(var)) = Ø	DEF(scanf(var)) = { var}
printf	REF(printf(var)) = { var}	DEF(printf(var)) = Ø
return	REF(return exp) = REF(exp)	DEF(return exp) = Ø
scanf 及 printf 外的函数调用	REF(函数名(实参)) = { 各实参表达式中的变量}	DEF(函数名(实参)) = Ø
break 和 continue	REF(break/continue) = Ø	DEF(break/continue) = Ø
谓词语句	REF(Predicate _ exp) = REF(exp)	DEF(Predicate _ exp) = Ø
entry 及 head 节点	REF(entry/head) = Ø	DEF(entry/head) = Ø

4.1.2 计算 GEN 和 KILL 集合

在获得 CDS 中各个节点的 REF 和 DEF 集合信息后，GEN 集合的生成将十分直接：在遍历 CDS 时，如果当前节点是赋值语句节点或 scanf 语句节点，则根据它的 DEF 集合，由当前的节点号和 DEF 中的变量名组成 GEN 集合中一个元素，从而建立 GEN 集合.

KILL 集合的生成需要对整个 CDS 中的节点进行多次遍历. 算法描述如图 4 所示.

```
算法: SetKill ( N)
输入: 已知各个节点的 DEF、REF 和 GEN 集合的控制依赖子图
输出: 求得各个节点的 KILL 集合的控制依赖子图
{
    1) 遍历控制依赖子图，将节点 N 的所有前驱节点，即所有在 N 之前执行的节点，加入到列表 LeftNodesList 中
    2) for( LeftNodesList 中的每个节点 P)
    {
      if ( P 是赋值语句节点或 scanf 语句节点)
      {
      检查 DEF [P] 中是否有 GEN [N] 中的变量，若有，假设为 x，则在 KILL [N] 中加入二元组 < x,P >
      }
      if ( P 是循环语句节点"ITERATION")
      {
      遍历节点 P 在控制依赖子图中的子图，检查其赋值语句子节点或 scanf 语句子节点 Q 的 DEF [Q] 中是否有 GEN [N] 中的变量，若有，假设为 x，则在 KILL [N] 中加入二元组 < x,Q >
      }
    }
}
```

图4 KILL 集合求解算法

4.1.3 计算 IN 和 OUT 集合

为了传播数据流信息，在计算到达－定值信息时需要使用 IN 和 OUT 集合. 一个节点的 IN 集合由到达该节点的前一点的定值组成，OUT 集合由到达该节点下一点的定值组成. 由于一个语句节点的 IN 和 OUT 集合可能不同，本文为 CDS 中的每个语句节点都计算这两个集合.

采用迭代法求各个节点的 IN 和 OUT 集合. 在每次迭代时对 CDS 中每个节点 N，使用 KILL 和 GEN 集合计算 IN 与 OUT 集合. IN [N] 等于节点 N 在 CDS 中前驱节点的 OUT 集合，OUT [N] 是用 IN [N]、GEN [N]、KILL [N] 生成的. 算法描述如图 5 所示.

到此为止，CDS 中每个节点的到达－定值信息就计算完毕了.

```
算法: Reaching-Definition
输入: 已知各个节点的 DEF、REF、GEN 和 KILL 集合的控制依赖子图
输出: 求得各个节点的 IN 和 OUT 集合即每个节点的到达－定值信息的控制依赖子图
{
  for( 控制依赖子图中的每个节点 N)
  {
    IN [N] = Ø
    OUT [N] = GEN [N]
  }
  change = true
  while( change)
  {
     change = false
    for ( 控制依赖子图中的每个节点 N)
    {
      IN [N] = OUT [P] //P 是 N 的前驱节点
      OLDOUT [N] = OUT [N]
      if ( N 是"ITERATION"或"SELECTOR")
      {
        OUT [N] = OUT [C] //C 是 N 的最右子节点
      }
      else if ( N 是"SELECTION"节点)
      {
        OUT [N] = ∪ OUT [C] //C 是 N 的子节点
      }
      else if ( N 是赋值语句或 scanf 语句节点)
      {
        OUT [N] = GEN [N] ∪ ( IN [N] - KILL [N])
      }
      else
        OUT [N] = IN [N]
      if( OUT [N] ! = OLDOUT [N])
        change = true
    }
  }
}
```

图5 到达－定值信息的求解算法

4.2 建立数据依赖边

在获得了各个节点的 IN 和 OUT 集合后，后根序遍历 CDS，利用各节点的 IN 集合和 REF 集合即可获得节点间的数据依赖关系. 具体的规则是如果当前节点的 REF 集合中某个变量名和 IN 集合中某个二元组中的变量名相同，则认为当前节点和 IN 集合同一个二元组中给出的节点之间存在流依赖关系，并将生成的流依赖关系放入表中存储. 建立流依赖边和声明依赖边的算法如图 6 所示.

```
算法: DataDependece
输入: 已求得每个节点的到达 - 定值信息的控制依赖子图
输出: 建立了数据依赖关系的系统依赖图
{
    for( 控制依赖子图中每个节点 N)
    {
      if( REF [N] 不空)
      {
        for( REF [N] 中的每个变量 x)
        {
          for ( IN [N] 中每个二元组 < y,M >)
          {
            if ( y = = x)
            {
              建立从 M 到 N 的流依赖边
            }
          }
        }
        if ( N 不是变量声明节点)
        {
          对 DEF [N] 中的变量 x,在控制依赖子图中查找其
          变量声明节点 M; 在 M 的声明依赖边集合中加入 N
        }
      }
    }
}
```

图 6　程序系统 DDS 建立算法

5　系统依赖图的生成实例

如图 7,8 中所示的两个示例程序，均实现相同的功能，但循环结构和选择结构采用了不同的语法表示形式，Horwitz 方法采用不同的形式表示这两个程序，而本文创建的 SDG 可将这两个语义等价程序表示成相同的形式，如图 9 所示，从而可以统一分析、理解这两个程序，降低了程序理解的复杂度. 在此基础上，可以进一步执行程序标准化转换，消除代码多样化.

此外，与 Horwitz 方法相比，本文方法直接基于控制依赖子图分析数据流信息，并且可实现按需的数据流分析，降低了数据流分析的复杂度，节省空间和时间，使之适合于分析大规模代码.

```
void main( )
{
  int sum;
  int i;
  sum = 0;
  i = 1;
  while ( i < = 10)
  {
  sum = add( sum, i) ;
  i = add( i,0) ;
  }
  printf( "% d" ,sum) ;
  printf( "% d" , i) ;
}

int add( int a, int b)
{
  if ( b = = 0)
   inc( &a) ;
  else
   a = a + b;
  return a;
}

void inc( int* x)
{
   * x = * x + 1;
}
```

图 7　示例程序 1

```
void main( )
{
  int sum;
  int i;
  sum = 0;
  for ( i = 1; i < = 10; i = add( i, 0) )
  {
   sum = add( sum, i) ;
  }
  printf( "% d" ,sum) ;
  printf( "% d" , i) ;
}

int add( int a, int b)
{
  switch ( b) {
  case 0:
    inc( &a) ;
    break;
  default:
     a = a + b; }
  return a;
}
void inc( int* x)
{
   * x = * x + 1;
}
```

图 8　与示例程序 1 语义等价的示例程序 2

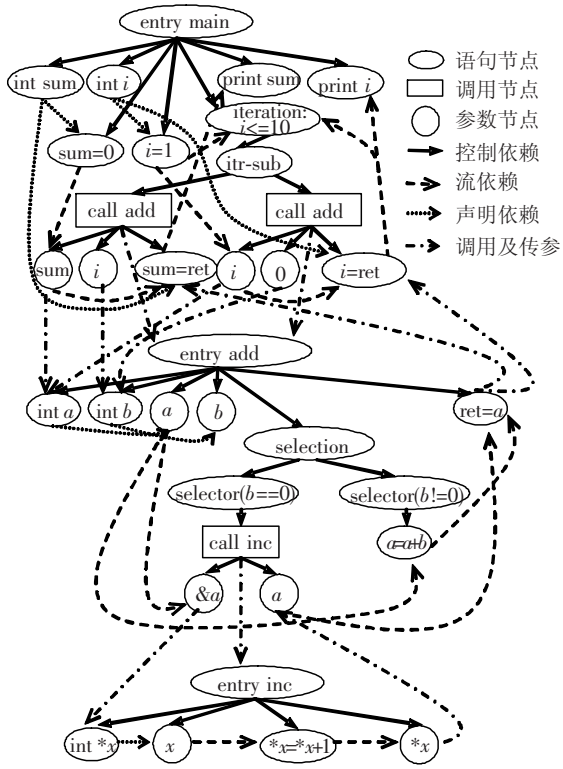

图 9　系统依赖图实例

6　SDG 在程序理解中的应用

本文提出的系统依赖图生成算法已在“基于程序理解的 C 语言编程题自动评分系统”中得到了应用，整个评分过程划分为 3 个阶段: 将程序代码转换成 SDG; 根据控制流和数据流对 SDG 进行标准化转换; 最后匹配标准化后的学生程序与模板程序控制依赖子图，并根据匹配结果给出评分. 由此可看到，编程题自动评分系统的基础和关键是 SDG.

本文算法还被应用于“语义相似的程序识别”中，识别大规模程序中语义相似的程序代码[19]. 其基本思想是: 分别将待检测的两段源代码解析为两棵系统依赖图的控制依赖树，并分别执行基本代码标准化; 利用度量值方法提取两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树; 对提取的候选相似代码，进行数据流分析，执行高级代

码标准化操作;计算语义相似度,获得相似度结果,从而识别语义相似的程序代码.

应用结果表明:本文构建的SDG能够充分表示程序的语法和语义,既可应用于分析小规模程序代码,也可应用于分析大规模程序代码,对可执行按需数据流分析,具有较低的复杂度和较高的准确性,并为程序标准化与语义分析提供了方便.

7 结 论

1)提出了一种面向程序理解的SDG构建算法,包含程序信息的提取、CDS的构建和DDS的构建3个阶段.

2)改进了传统的SDG,直接基于CDS分析数据流信息,统一表示选择语句和循环语句,将表达式表示为抽象语法树.

3)提出的SDG构建算法已被应用于基于程序理解的编程题自动评分系统和大规模程序的语义相似程序代码识别中,结果表明该SDG能够充分表示程序的语法和语义,且按需进行数据流分析的方法降低了程序理解和分析的复杂度.

参考文献

[1] EXTON C. Constructivism and program comprehension strategies [C]//Proceedings of the 10th International Workshop on Program Comprehension (IWPC′02). Washington, DC: IEEE Computer Society, 2002: 282-283.

[2] OTTENSTEIN K J, LOTTENSTEIN M. The program dependence graph in a software development environment [C]//Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. New York: ACM, 1984: 177-184.

[3] HORWITZ S, REPS T, BINKLEY D. Interprocedural slicing using dependence graphs [J]. ACM Transactions on Programming Languages and Systems, 1990, 12(1): 26-60.

[4] MATSUMOTO T, SAITO P, FUJITA P. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs [C]//Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED′06). Washington, DC: IEEE Computer Society, 2006: 370-375.

[5] 逄龙,王甜甜,苏小红,等. 支持多程序语言的静态信息提取方法 [J]. 哈尔滨工业大学学报, 2011, 43(3): 62-66.

[6] SRIDHARAN M, FINK P J, BODIK R. Thin slicing [C]//Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2007: 112-122.

[7] ANDERSON P, ZARINS M. The codesurfer software understanding platform [C]//Proceedings of the 13th International Workshop on Program Comprehension. Washington, DC: IEEE Computer Society, 2005: 147-148.

[8] MILLER J, REFORMAT M, ZHANG H. Automatic test data generation using genetic algorithm and program dependence graphs [J]. Information and Software Technology, 2006, 48(7): 586-605.

[9] WANG Tiantian, SU Xiaohong, MA Peijun, *et al*. Ability-training-oriented automated assessment in introductory programming course [J]. Computers & Education, 2011, 56(1): 220-226.

[10] WANG Tiantian, SU Xiaohong, WANG Yuying, *et al*. Semantic similarity-based grading of student programs [J]. Information and Software Technology, 2007, 49(2): 99-107.

[11] 马培军,王甜甜,苏小红. 基于程序理解的编程题自动评分方法 [J]. 计算机研究与发展, 2009, 46(7): 1136-1142.

[12] 杨轶,苏璞睿,应凌云,等. 基于行为依赖特征的恶意代码相似性比较方法 [J]. 软件学报, 2011, 22(10): 2438-2453.

[13] WANG Tiantian, SU Xiaohong, MA Peijun. Program normalization for removing code variations [C]//International Conference on Computer Science and Software Engineering. Washington, DC: IEEE Computer Society, 2008: 306-309.

[14] SNELTING G, ROBSCHINK T, KRINKE J. Efficient path conditions in dependence graphs for software safety analysis [J]. ACM Transactions on Software Engineering and Methodology, 2006, 15(4): 410-457.

[15] 王甜甜,苏小红,马培军. 程序标准化转换中的指针分析算法研究 [J]. 电子学报, 2009.37(5): 1104-1108.

[16] DU Lin, XIAO Guorong, LI Daming. A novel approach to construct object-oriented system dependence graph and algorithm design [J]. Journal of Software, 2012, 7(1): 133-170.

[17] ZHAO Jianjun, RINARD M. System Dependence Graph Construction for Aspect-Oriented Programs [R]. MIT-LCS-TR-891, MIT: Laboratory for Computer Science, 2003.

[18] SILVA J TAMARIT S, TOMÁS C. System dependence graphs in sequential erlang [C]//Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering. Washington, DC: IEEE Computer Society, 2012: 486-500.

[19] 王甜甜,马培军,苏小红,等. 一种基于程序源代码语义分析的代码相似度检测方法:中国, 200910073094.4 [P]. 2009.

(编辑 张 红)