一种C++程序切片系统的设计与实现

周 婕,慕晓冬,王 杰
(第二炮兵工程学院,陕西 西安 710025)

摘 要:程序切片是一种程序分析技术,它主要应用在程序的调试和逆向工程。文中介绍了笔者设计并实现的一个C++程序切片系统。其中包括系统的总体框架,系统中使用的数据结构以及切片生成算法。本系统已经成功应用到了X型导弹指挥软件的故障诊断中。

关键词:程序切片;程序依赖图;抽象语法树
中图分类号:TP311.1 **文献标识码**:A **文章编号**:1673-629X(2006)07-0020-03

Design and Implementation of a Kind of C++ Program Slicing

ZHOU Jie, MU Xiao-dong, WANG Jie
(The Second Artillery Engineering College, Xi'an 710025, China)

Abstract: Program slicing is a program analysis technique. It is mainly used during debugging and reengineering. Present a C++ program slicing system including the general idea of design, data structures and slicing algorithm used in the system. The system has been applied in software fault diagnosis.

Key words: program slicing; program dependence graph; AST

切片是一种程序分析技术,其最初的概念是由Mark Weiser于1979年在他的博士论文中提出来的。Weiser定义的切片是通过从源程序中移去零条或多条语句而得到的一个可执行程序。切片准则是指程序中的某个兴趣点与在这个兴趣点上定义和使用的变量组成的二元组。通常计算的切片就是在源程序中影响切片准则和受该切片准则影响的语句的集合[1]。

1 系统的总体框架

利用开放编译器GCC(GNU COMPILER COLLECTION)的前端对源程序进行词法和语法分析[2],并得到抽象语法树;进一步优化填充全局符号表;基于全局符号表构造依赖图,基于依赖图计算相关的程序切片,并存放到切片库中,得到程序切片[3]。

在具体的实现方案中,关键是要为C++源程序构造出依赖图这种中间表示形式。由于依赖图实际由控制依赖子图和数据依赖子图构成,所以在构造依赖图时,只要构造程序调用图、控制依赖子图和数据依赖子图就可以了[4]。首先构造依赖图以提取源程序的信息,具体的框架如图1所示,下面对方案图中的每一部分做简单的介绍。

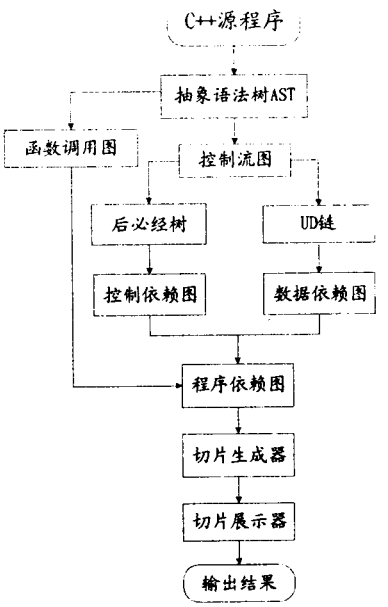

图1 系统的总体框架

(1)构造程序调用图时,直接从抽象语法树(AST)中提取函数的先后调用信息。

(2)控制依赖子图是语句级依赖图的一部分。该子图表示的是一个语句或表达式执行时依赖的控制条件。首先将从抽象语法树(AST)转换成控制流图G,再由G构造后必经结点树T,然后在T上根据后必经关系,寻找语句间的控制依赖关系。

(3)数据依赖子图同控制依赖子图一样,也是语句级

收稿日期:2005-10-05
作者简介:周 婕(1978-),女,浙江宁波人,硕士研究生,主要研究方向为软件工程、数据库应用、智能化信息处理等;慕晓冬,教授,博士生导师,主要研究方向为多媒体技术、软件工程、信息融合等。

依赖图的一部分,它也是基于控制流图来构造的。数据流分析沿着所有的控制路径,把到达一定值信息存储为引用—定义链(UD链),它是所有能够到达变量的某个引用的定值表,然后根据UD链求取语句间的数据依赖关系。

(4)子程序间切片是基于改进的程序依赖图进行的。

2　系统的具体设计

2.1　从源程序获得抽象语法树

在对C++源程序进行程序分析前(数据流分析、控制流分析),通常需要对源程序进行词法、语法分析。这些分析从源程序中提取信息,将源程序转换成其相应的内存形式(中间表示),以便后续分析,可以方便地查询信息,提高查询速度,减少查询的代价。本系统选择抽象语法树(AST)作为中间表示。通常把源程序对应的AST作为一种树结构,能够比较直观地表示出源程序语言的语法结构,具有较高的存储效率,遍历和操作AST十分方便。

2.2 根据抽象语法树AST填充全局符号表

2.2.1　全局符号表概念

在切片生成过程中,需要对程序进行分析与处理,提取一些重要信息,诸如,变量属性信息、语句信息、函数单元结点信息、类信息等,将这些有用的信息存入适当数据结构中,这种数据结构就是全局符号表。

2.2.2　全局符号表的数据结构

(1)类层符号表。

该层符号表记录单个类的信息,包括类的id、类名、类的继承信息以及类中所有成员的信息等。其符号表定义如下:

```
typedef struct classdef{
  int classid_ch; /*类的 id */
  char *classname_ch; /*类名*/
  name_list_ch *inheritable_ch; /*基类的名字列表*/
  memberdef_ch *membertable_ch;/*记录一个类中所有成员变量和成员方法的信息,用单链表表示*/
  } classed_ch;
```

(2)成员层符号表。

该层符号表记录类中一个成员(成员变量或成员方法)的信息,其符号表定义如下:

```
typedef struct memberdef {
  int memberid_ch; /*成员变量或成员方法的 id */
  int is_membervar_ch; /*若表示一个成员变量,则为1;否则为0*/
  char *membername_ch; /*成员变量名或成员方法名*/
  name_list_ch *datatype_ch;/*成员变量的类型或成员方法所有形参的类型*/
  name_list_ch *create_classname_ch;/*记录一个成员方法中创建的所有类的类名,用于构造创建关系*/
  name_list_ch *callmethod_ch;/*记录一个成员方法中调用的方法名*/
  IN_ch *var use_ch;/*成员方法中引用的变量*/
  OUT_ch *var_def_ch;/*成员方法中修改的变量*/
  stmtdef_ch *explain_ch;/*成员方法中说明性变量*/
  int entry_node; /*函数的输入结点编号*/
  int exit_node; /*函数的输出结点编号*/
  } memberdef_ch;
```

(3)语句层符号表。

程序依赖性分析是以语句为单位的,故对程序P要建立一张语句表来存放有关信息,程序中的每一个语句将在表中占一行。

```
typedef struct stmtdef{
  int stmtid_ch:/*语句的语句号*/
  enum stmtkind:/*语句类型*/
  int sin;/*指向语句中引用变量在IN表中的入口位置*/
  int sout; /*指向语句中定义变量在OUT表中的入口位置*/
  into obrother; /*表示该语句的兄语句号*/
  int ybrother; /*表示该语句的弟语句号*/
  stmtdef_ch *pred; /*指该语句的直接前驱语句链*/
  stmtdef_ch * succ; /*指该语句的直接后继语句链*/
  } stmtdef_ch;
```

(4)变量说明符号表。

下面对IN表和OUT表分别进行说明。

```
typedef struct IN{
  name_list_ch *varname_ch;/*变量名*/
  int usei;/*引用变量所在的语句号*/
  int explain;/*该变量说明处的语句号*/
  int memberno;/*语句的函数单元号*/
  }IN_ch;
typedef struct OUT{
  name_list_ch *varname_ch;/* 变量名*/
  int defi;/*被修改的变量所在语句号*/
  int explain;/*该变量说明处的语句号*/
  int memberno;/*语句的函数单元号*/
  }OUT_ch;
```

2.3　程序依赖图

2.3.1　定义

定义1　程序P的程序依赖图(PDG)是一个有向图,可用二元组(S',E')表示。其中,S'为节点集,$S'=S$(S为P的CFG中的节点集);边集E'表示节点间的依赖关系。即$<S1,S2>\in E'$表示CD($S1$,$S2$)或者DD($S1$、$S2$)[1]。

本系统是基于改进的程序依赖图,下面给出改进程序依赖图的定义[5]。

定义2　改进的程序依赖图(TPDG)是一个带标记的有向图,可用三元组(S', E', T)表示,节点集$S'=S$(S与P的控制流图的节点一一对应),边集$E'=E1\cup E2$,其中:

$E1=\{<S1,S2>|\mathrm{CD}(S1,S2)\}$为直接控制依赖边;

$E2=\{<S1,S2>|(x,S2,y)\in \mathrm{Dep\_}D(S1)\ \vee$

$(x, S2, x) \in Dep\_R(S1)\}$ 为数据依赖边，并且由不同变量引起的依赖关系对应不同的边(因为它们的标记不同)。每一条边都有形如(x,y)的标记(x, y 为变量或者"*")。

边 $e = < S1, S2 >$ 上的标记由如下方法获得：

· 若 $e \in E1$，则 e 的标记为(*，*)；

· 若$(x, S2, y) \in Dep_D(S1)$，则 e 的标记为(y, x)；

· 若$(x, S2, x) \in Dep_R(S1)$，则 e 的标记为(x, x)。

其中$Dep_D(S)$为数据定义依赖集，$Dep_R(S)$为数据引用依赖集。它们的定义如下：

定义 3 $Dep\_D(S) = \{(x, t, y) \mid x \in Def(S) \wedge y \in Def(t) \wedge y \in Def(S,x) \wedge (t, y) \in ln(S)\}$。若$(x, t, y) \in Dep_D(S)$，则称 S 中定义的变量 x 定义依赖于 t 中定义的变量 y。

定义 4 若 S 为一控制语句(如 if, while)，x 为控制条件中引用的变量，则 $Dep\_R(s) = \{(x, t, x) \mid x \in Ref(S) \wedge x \notin Def(S) \wedge x \in Def(t) \wedge (t, x) \in ln(S)\}$；若$(x, t, x) \in Dep_D(S)$，则称 S 中引用的变量 x 引用依赖于 t 中定义的变量 y。

2.3.2 程序依赖图的数据结构

(1)依赖图的结点。

依赖图中的结点是用来存储程序中一条一条的语句，一条语句对应于一个结点。

```
typedef struct DepNodeType{
    int stm_id;/*语句编号*/
    int def_exp; /*修改变量编号*/
    int use_set; /*引用变量编号*/
    int post_dominator_node;/*后向必经结点*/
    int control_dep_edges;/*属于控制依赖边上结 点*/
    int data_dep_edges; /*属于数据依赖边上结点*/
    enum type; /*结点类型*/
    bool visited;/*用来标记结点是否被遍历*/
} DepNodeType;/*依赖图的结点*/
```

(2)依赖图的边。

依赖图的边是用来存储语句和语句之间的关系，是数据依赖还是控制依赖。

```
typedef struct DepEdgeType{
    int target_node;/*目标语句结点号*/
    int next;/*下一个语句结点号*/
    int Label[2];/*用来标记结点之间的关系，是控制依赖，引用依赖还是定义依赖*/
} DepEdgeType; /*依赖图的边*/
```

3 子程序间切片算法

如果一个语句包含子程序调用语句，首先将其看作一般的语句，并对调用程序进行子程序内切片；然后对被调用子程序作相应的切片分析。其主要工作是首先确定被调用子程序的切片准则，这样，子程序间的切片就转化为子程序内的切片[6]。

给定切片准则 $< s, v >$，子程序间的切片可通过子程序间切片算法获得。在此算法中，每个子程序单独被切片，CallStack 是个集合，它记录了被调用的子程序，其中每个元素的形式为(P,x)，P 为子程序名，x 是 P 的形式参数，如果 P 是个函数，要对其返回值做切片，此时它的形式为(P,P)。算法的前半部分计算子程序内的切片。在该算法中，除第一次需分析所有子程序切片外，其它切片分析时可直接重用记录在 SliceList[s,v] 中的以前计算的结果，只需计算切片标准中语句所在的子程序。这样，切片分析的效率将大大提高。

子程序间切片算法如下：

```
输入：程序 P 的程序依赖图和切片准则〈s,v〉
输出：程序切片 Pslice，相关变量 Calleevars
全局变量：CallStack：程序 P 的函数调用图
          SliceList[s,v]：
              SliceList[s,v].entryvars 和
              SliceList[s,v].pslice 存储切片信息
声明：WorrKList 用来存储 P 的结点列表
      Def[s,v]和 Rel[s,v]来存储定义和引用结点
算法 ComputePSlice：
if SliceLIST[s,v]≠NULL then
  对 SliceLIST[s,v].entryvars 进行 ComputePSlice
else
  初始化各个数据结构：
    WorKList = ∅,Pslice = ∅,Callstack = ∅
  for all v∈Def[s,v] ∨ Rel[s,v] do
    WorKList = WorKList ∪ {(s,v)}
  end for
repeat
  while WorKList≠∅ do
    从 WorKList 中取出一个元素〈s,x〉
    for 每个未标记的边〈s,s′〉do
      标记<s,s′>，设 T 为边〈s,s′〉上的标记
    if T = <y,x> then//数据依赖
      WorKList = WorKList ∪ {(s,v)}
      Pslice = Pslice ∪ {s′}
    if y 为函数名 then
      将(y,y′)加入到 CallStack
      SliceList[s,v].entryvars = (s′,y)
      SliceList[s,v].pslice = Pslice
    if y 为子程序 R 的实参
    then 设 y′为相应的形参
      将(R,Y,)加入到 CallStack
      SliceList[s,v].entryvars = (s′,y)
      SliceList[s,v].pslice = Pslice
  if T = < * , * > then//控制依赖
      Pslice = Pslice ∪ {(s′,y)
```

(下转第 25 页)

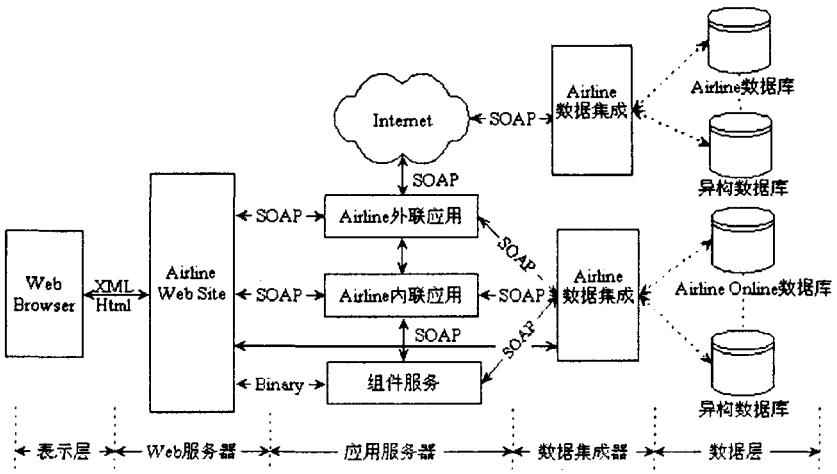

图 3　基于 Web 服务的在线航班预订系统体系结构

Airline 网站,实现客户的登录注册航班查询机票预订订单生成交易查询和旅行信息浏览等功能。Airline 内联应用,负责与内部网络数据集成的 Web 服务进行交互,同时实现系统的内部应用逻辑。Airline 数据集成,是对数据库进行集成,屏蔽了后台数据库的异构性,对外提供统一的 XML 数据。Airline 系统的实现基于上述的电子商务应用体系结构及数据集成方案,较好地满足了电子商务系统的应用需求,具有良好的信息交换及容易扩展和部署等优点,并且可以跨平台,穿透 Firewall 调用和访问。如完成以查询 2004 年 9 月 10 日起飞,2004 年 9 月 11 日抵达,由北京飞往 Ottawa 的航班查询核心 XML 文档如下:

```
<Flight diffgr :id ="Flight1"msdata :rowOrder ="0">
<flight id>0001</flight id>
<flight number>1000</flight number>
<startfrom>北京</startfrom>
<destination>Ottawa</destination>
<departure date>2004 - 09 - 11T10:20:00.0000000 + 08:00</departure date>
<arrival date>2004 - 09 - 12T18:45:00.0000000 + 08:00</arrival date>
<price>18,000.00</price>
```

4　结束语

文中给出的基于 Web Service 的体系结构在实践应用中取得了良好的效果,但因为 Web Service 技术不成熟等,在安全性、服务质量和管理等方面需要对基于 Web Service 的电子商务系统进行改进。

参考文献:

[1] Huang S, Yen D, Kwan I, et al. Developing and XML gateway for business - to - business commerce [A]. Proceeding of IDS2000[C]. Hong Kong: [s. n.], 2000.

[2] Donox, Ehnebuske D, Kakivaya G. Simple Object Access Protocol(SOAP)[S]. 2001.

[3] Liu Jianxun, Zhang Shensheng. An Inter - enterprise Workflow Model for Supply Chain and B2B E - commerce[J]. High Technology Letters, 2002, 8(2): 51 - 55.

[4] Linthicum D S. B2B Application Integration: E - Business Enable Your Enterprise[M]. [s. l.]: Addison - Wesley, 2001. 201 - 220.

[5] W3C. Web Services Description Language (WSDL) 1.1[S]. 2001.

(上接第 22 页)

```
    for 每个 yi∈Ref(s')do
      WorKList = WorKList∪{(s',y)}
    if y 为函数名 then
    将(y,y)加入到 CallStack
    SliceList[s,v].entryvars = (s',y)
    SliceList[s,v].pslice = Pslice
until WorKList = NULL
while CallStack≠∅ do
  从 CallStack 中取出一个元素(R,Y)
  对子程序 R 进行关于参数 Y 的切片
  设 n 为子程序 R 的输出结点
  SliceList[R,y].entryvars = (n,y)
  SliceList[R,y].pslice = 关于参数 y 的切片
  Pslice = Pslice∪关于参数 y 的切片
  if 程序 R 中包含有程序调用语句
  then CallStack = CallStack∪(Q,z)
```

4　结　论

文中所设计的 C++程序切片系统主要是用于 C++程序的排错和故障诊断。本系统已经应用到了 X 型导弹指挥软件的故障诊断中,缩小了故障诊断的范围,提高了诊断的针对性和诊断的效率。

参考文献:

[1] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization[J]. ACM Transactions on Programming Languages and Systems, 1987, 9(3): 319 - 349.

[2] GCC Home Page[EB/OL]. http: //gcc.gnu.org, 2005 - 07.

[3] 陈意云,马万里.编译原理和技术[M].合肥:中国科学技术大学出版社,1989.

[4] 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.

[5] 徐宝文.一种逆向程序流依赖性分析方法及其应用[J].计算机学报,1993,16(5):385 - 392.

[6] Horwitz S. Interprocedural Slicing Using Dependency Graphs [J]. ACM Trans Programming Languages and Systems, 1990, 12(1): 26 - 60.