ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.4, April 2007, pp.832−841
DOI: 10.1360/jos180832

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种基于异常传播分析的依赖性分析方法*

姜淑娟[1,2,3+], 徐宝文[1,2], 史 亮[1,2], 周晓宇[1,2]

[1](东南大学 计算机科学与工程学院,江苏 南京 210096)

[2](江苏省软件质量研究所,江苏 南京 210096)

[3](中国矿业大学 计算机科学与技术学院,江苏 徐州 221008)

An Approach to Analyzing Dependence Based on Exception Propagation Analysis

JIANG Shu-Juan[1,2,3+], XU Bao-Wen[1,2], SHI Liang[1,2], ZHOU Xiao-Yu[1,2]

[1](School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

[2](Jiangsu Institute of Software Quality, Nanjing 210096, China)

[3](School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221008, China)

+ Corresponding author: Phn: +86-516-83591721, E-mail: shjjiang@cumt.edu.cn

Jiang SJ, Xu BW, Shi L, Zhou XY. An approach to analyzing dependence based on exception propagation analysis. *Journal of Software*, 2007,18(4):832−841. http://www.jos.org.cn/1000-9825/18/832.htm

Abstract: Based on analyzing the effects of exception handling constructs on dependence analysis, this paper proposes a precise and efficient representation for C++ programs with exception handling constructs—improved control flow graph. It proposes a new approach to analyzing the data dependences and control dependences of intra-function and inter-function in C++ programs with exception handling constructs, and an efficient algorithm is also presented. This method overcomes the limitations of the previous incorrect analysis because of failing to account for the effects of exception handling constructs, and also provides a basis for automatic dependence analysis that contains exception propagation. Finally, it discusses the application of the dependence analysis method in program slicing.

Key words: exception handling; program analysis; exception propagation; dependence analysis; program slicing; robustness

摘 要: 在分析异常处理结构对程序依赖性分析影响的基础上,对传统的控制流图进行改进,提出了一种新的能够描述包括异常处理结构在内的函数内和函数间的 C++程序的依赖性分析模型,并给出了相应的构造算法.该方法

* Supported by the National Natural Science Foundation of China under Grant No.60373066 (国家自然科学基金); the National Science Foundation for Distinguished Young Scholars of China under Grant No.60425206 (国家杰出青年科学基金); the National Grand Fundamental Research 973 Program of China under Grant No.2002CB312000 (国家重点基础研究发展规划(973)); the Program for Cross-Century Outstanding Teachers of the Ministry of Education of China (国家教育部跨世纪优秀人才基金); the High Technology Research Project of Jiangsu Province of China under Grant No.BG2005032 (江苏省高技术研究项目); the Natural Science Foundation of Jiangsu Province of China under Grant No.BK2006094 (江苏省自然科学基金); the Science Research Foundation of China University of Mining and Technology under Grant No.OD4527 (中国矿业大学科学研究基金)

Received 2005-12-21; Accepted 2006-08-16

既克服了因忽略异常处理结构对程序依赖性的影响而造成分析结果不准确的不足,又有助于实现基于异常传播的程序依赖性分析的自动处理.最后,对这种依赖性分析方法在程序切片中的应用作了进一步的探讨.

关键词: 异常处理;程序分析;异常传播;依赖性分析;程序切片;健壮性
中图法分类号: TP311 文献标识码: A

程序依赖性分析是进行程序分析、软件理解与软件维护所必不可少的工作,在软件维护、程序测试、优化和并行性分析等方面有着广泛的应用[1−3].依赖性分析方法一般分为两种:一种是沿着程序执行流由前至后的正向分析方法,一般用来分析整个程序的依赖关系;另一种是由程序中某个特定点开始逆着执行流由后向前的逆向分析方法,主要用于计算程序中这一特定点的依赖关系[3].子程序间依赖性分析是程序依赖性分析的重点和难点,人们提出了很多方法,如 Horwitz 等人提出了系统依赖图(system dependence graph,简称 SDG)概念,并提供了一种机制来跟踪被调用子程序的调用上下文[4],但这种方法并没有考虑到异常处理结构对程序依赖性的影响,如果所分析的程序中包含异常处理结构,则采用这种方法得出的结果是不准确的.然而,现代高级程序设计语言如 Ada,C++,Java 等都支持异常处理机制,而且使用异常处理机制来开发健壮的程序已越来越普及[5],因此,在进行程序的依赖性分析时必须考虑异常处理结构对其造成的影响.

目前对程序的依赖性分析大多集中在 Java 语言[5−9].异常的传播是异常处理结构影响程序依赖性分析的一个主要方面.在 Java 语言中,对于显式抛出的异常,如果本方法中没有异常处理程序与之匹配,则编译器会指出有语法错误.Java 的这些限制极大地简化了 Java 程序中由于异常传播所引起的依赖性分析的复杂程度.但 C++语言对于异常的传播比较灵活,这种灵活性增加了程序中依赖性分析的复杂程度.把对 Java 程序的依赖性分析方法应用到 C++程序的依赖性分析中,还有许多需要修改的地方.

我们曾对程序依赖性分析做过比较深入的研究,取得了一定的成果[1−3,10,11].本文是在以前工作的基础上,扩展传统的依赖性分析方法来适应含有异常处理结构的 C++程序的依赖性分析,并给出相应的算法.

1 异常处理结构对程序依赖性的影响

当一个程序在执行过程中引发一个异常时,异常处理机制负责使程序的正常控制流改变到一个异常的控制流(即转到一个异常处理程序).异常的引发改变了程序的正常控制流,使程序沿着异常的控制流执行,从而引起了控制依赖的变化.如 C++语言中的 throw 语句,就如同一个 goto 语句,使程序直接转到异常处理程序处执行.同样,程序的控制流发生变化必然会引起程序中某些变量的定值——引用链的变化,从而使数据的依赖性发生变化.

在顺序程序中,数据的依赖关系主要包括由控制条件和函数调用引起的控制依赖、由访问变量和参数传递引起的数据依赖以及由于异常的引发和传播而引起的控制依赖和数据依赖[4,5,9].

1.1 异常处理结构对控制依赖分析的影响

控制流图 CFG 是描述程序控制流的一种图示方法,它反映了程序中语句的执行顺序和函数之间的相互调用关系[4].许多程序分析技术和软件工程任务都使用控制流图来表示程序的信息.语句层次上的控制流图表示程序所有可能的执行路径,控制流图中的结点表示程序语句,边表示程序语句之间可能的控制转移.

定义 **1**. 设 s_1,s_2 为控制流图中两个节点,s_2 能否被执行由 s_1 的执行状态决定,则称 s_2 控制依赖于 s_1.

一般情况下,复合语句(如 if,while)内部的语句控制直接依赖于其〈条件部分〉.节点间的控制依赖可通过分析源程序的控制语句直接获得.

为了得到准确的控制流信息,需要把异常处理结构引入到语句层次的控制流图中,即在分析程序的控制流时考虑执行 throw 语句、catch 语句的所有路径,它在函数内的控制流图中增加了下面的路径:

① try 块抛出异常且该层的 catch 块捕获了异常,正常退出;

② try 块抛出异常,该层也找到了匹配的处理程序,但在 catch 块又重新抛出了异常或又引发其他异常,异

常将向外层的 try 块(如果有 try 块的嵌套)传播或沿调用栈传播;

③ try 块抛出异常但该层没有找到匹配的处理程序,异常将向外层的 try 块(如果有 try 块的嵌套)传播或沿调用栈传播.

异常处理机制在函数间的控制流图中增加了下面的路径:

① 被调用方法传播了异常但调用方法捕获了异常;

② 被调用方法传播了异常但调用方法中没有匹配的处理异常,异常将沿调用栈继续传播.

1.2 异常处理结构对数据依赖分析的影响

程序中的依赖关系除了语句之间的控制依赖以外,更多涉及到的是变量之间的数据依赖.

定义 **2**. 设 s_1 与 s_2 为控制流图中的两个节点,v 为一个变量,如果:

① s_1 对变量 v 进行了定义;

② s_2 执行时使用了 v 的值;

③ s_1 与 s_2 之间存在一条可执行路径,且在此路径上没有语句对 v 进行重新定义.

若满足这 3 个条件,则称 s_2 关于变量 v 数据直接依赖于 s_1.

数据流分析就是收集程序中使用数据、定义数据和数据依赖等信息的过程.目前有两种基本方法分析含有异常处理结构的数据流:一种方法是在数据流中增加异常处理结构的数据流;另一种方法是把异常处理结构的数据流单独表示出来[12].我们采用了前一种方法,即在讨论异常处理结构对数据依赖分析的影响时,用控制流图来表示程序信息,并在原数据依赖图中增加由于异常处理结构的引入所引起的数据依赖边.这样,从控制流图中就可以很容易地计算出包含异常处理结构程序的数据依赖.

1.3 一个例子

下面用图 1 中的例子来说明异常处理结构对程序依赖性的影响.

```
1        #include 〈iostream.h〉
2        class Ex1 {};
3        class Ex2: public Ex1 {};
4          int s;
5 +*     int f(int a, int b)
6          {int c;
7 +*        if (b==0) throw Ex1();
8   *            c=a/b;
9   *             s=111;
10 *             return c;
11       }
12 +*   void m(int x, int y)
13       {int z=7; s=0;
14 +*         try{
15 +*             z=f(x,y);
16                cout<<"z1="<<z;
17 +*            }
18 +*          catch (Ex2 & e2) {
19   *                    cout<<"Ex2 occurred s="<<s; }
20 +*          catch (Ex1 & e1) {
21 +*                    cout<<"Ex1 occurred s="<<s; }
22            cout<<"s="<<s;}
```

Fig.1 An example of dependence analysis

图 1 依赖性分析实例

在图 1 中,函数 *m*()调用函数 *f*(),函数 *f*()可能引发异常类型为 Ex1 的异常.由于函数 *f*()中没有提供相应的异常处理程序,所以,异常会传播到函数 *m*()内.首先检查控制依赖.第 21 行语句的执行与否依赖于第 7 行的 throw 语句的执行,第 16 行语句的执行与否依赖于第 15 行的函数调用语句是否能够正确返回:如果能够正确返回,则

第 16 行的语句执行;否则,第 16 行的语句根本不执行.

再来检查数据依赖.对于变量 s 来说,由于 s 是全局变量并且在被调用函数中重新赋值,所以,第 22 行的输出语句中 s 的值受到第 15 行的函数调用语句能否正确返回的影响,即依赖于在函数 $f()$中是否引发异常.如果在 $f()$中没有引发异常并正确返回,则第 22 行的语句中 s 的值与第 9 行的 s 的值相同;如果在 $f()$引发了异常,则第 22 行语句中 s 的值为第 13 行中的 s 的值,与第 21 行中 s 的值相同.

2 函数内依赖性分析

根据全局变量在函数中的使用情况,将其作为 in,out 或 in out 参数处理.为了描述函数间的参数传递过程,引入 5 种新的节点[11]:formal-in,formal-out,actual-in,actual-out 和 call-site.如果某参数是只读的,则称为 in 类型参数;如果某参数是只写的,则称为 out 类型参数;如果某参数是可读可写的,则称为 in out 类型参数.在引入异常处理机制之前,影响控制流的结构元素有:断言、goto 语句、break 语句和 return 语句等;引入异常处理机制以后,影响控制流的结构元素又增加了 throw 语句和 catch 语句.

定义 **3**. 程序 P 的程序依赖图(program dependence graph,简称 PDG){XE "PDG:程序依赖图"}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$\langle s_1,s_2\rangle\in E'$表示节点 s_2 控制依赖于 s_1 或数据依赖于 s_1.

在顺序程序中,函数间的数据依赖和控制依赖关系具有传递性,即:设 s_1,s_2 与 s_3 是函数 P 的 3 个语句,如果 s_2 依赖于 s_1 且 s_3 依赖于 s_2,则 s_3 依赖于 s_1.

大多依赖性分析算法建立在程序依赖图的基础上[1−4],它表示语句之间的依赖关系.程序依赖图的构建以控制流图为基础.扫描 C++程序源代码,对于程序中的每一个函数生成一个改进控制流图 ICFG,构造方法如下:

一个函数可能有几个结束语句(或出口),为了保证控制流图中终点的唯一性,我们在图中加入一个虚节点 exit,并将所有结束语句对应的节点都指向它.与通常构建控制流图的方法相比,有下列不同:

① 对每一个能抛出异常的函数,除了通常的 exit 节点外,CFG 增加了一个 norm_exit 节点和一个 excep_exit 节点.对每一种可能抛出的异常类型都有一个带有异常类型的 exit 节点,其后继节点是 excep_exit 节点.norm_exit 节点跟在函数中的最后一个语句之后,函数的 exit 节点是 norm_exit 和 excep_exit 的后继节点.

② 对一个函数调用节点 callsite,可能有两种情况:一种情况是函数正常返回;另一种情况是被调用函数引发了异常,异常返回.对每一个函数调用节点,增加一个 normal-return 节点.对每一个函数调用节点有一个标记为“T”的出边连到 normal_return 节点,另一条标记为“F”的出边连到相应的 catch 边(如果 callsite 在 try 块内),否则连到当前函数的 excep_exit 节点.normal_return 节点的出边连到当调用正常返回时执行的第 1 个语句,如果 callsite 是函数中的最后一条语句,则连到该函数的 norm_exit 节点.

③ 对每一个 try 语句,有一条出边连到 try 块内的第 1 个语句.

④ 对每一个 catch 语句,有一条标记为“T”的出边连到它们的第 1 个语句,另一条标记为“F”的出边连到与 try 匹配的这组 catch 块中的下一个 catch 语句(如果有的话),或上一层的 catch 语句(如果有 try 块嵌套),否则连到该函数的 excep_exit 节点.在 Java 程序中,如果在 try 块中显式引发了异常,而在 catch 块中并没有定义与之匹配的处理程序,则编译器会指出有语法错误,而在 C++程序中编译器并没有这样的检查功能.所以,在 C++程序中会出现在 try 块中显式引发了异常,而在 catch 块中并没有定义与之匹配的处理程序的情况.如果是 catch(…)节点,则没有“F”边.

⑤ 对每一个在 try 块内的 throw 语句,有一条标记为“T”的出边连到相应的 catch 节点;对每一个不在 try 块内的 throw 语句,或在异常处理程序中的 throw 语句,有一条标记为“T”的出边连到该函数的 excep_exit 节点.无论 throw 语句是否在 try 块内,都有一条标记为“F”的出边连到紧跟在 throw 语句后的第 1 个语句(如果不抛出异常而执行的语句).

⑥ 为了处理数据依赖,我们对可能抛出异常的每一个函数的 formal-out 型的形参,增加一组 $v\_out=v$ 节点作为每个函数的 norm_exit 节点的后继;对调用可能抛出异常的函数的每一个调用点的 actual-out 型的实参,增

加一组 *v=v_out* 节点作为 normal_return 节点的后继节点.由于异常退出不会产生函数返回值,所以,在 excep_exit 之后不必添加 *v_out=v* 节点.对于全局变量,在函数的入口处增加 *v=v_in* 节点,在函数的出口处(在 norm_exit 节点和 excep_exit 节点处)增加 *v_out=v* 节点;对可能抛出异常的被调用函数中用到的每一个全局变量,增加一组 *v=v_out* 节点作为 normal_return 节点和 catch 节点(exception return 节点)的后继节点.图 2 是图 1 中 C++程序的改进控制流图 ICFG.

在程序依赖图中,全局变量作为函数的参数.除 exit 节点外,PDG 中的节点与 ICFG 中的节点完全相同;PDG 中的边表示数据依赖和控制依赖.formal-in 节点和 formal-out 节点控制依赖于所在函数的入口节点;actual-in 节点和 actual-out 节点控制依赖于调用点 call-site.根据上一节关于控制依赖和数据依赖的定义,增加相应的依赖边,即可完成 PDG 的构建.

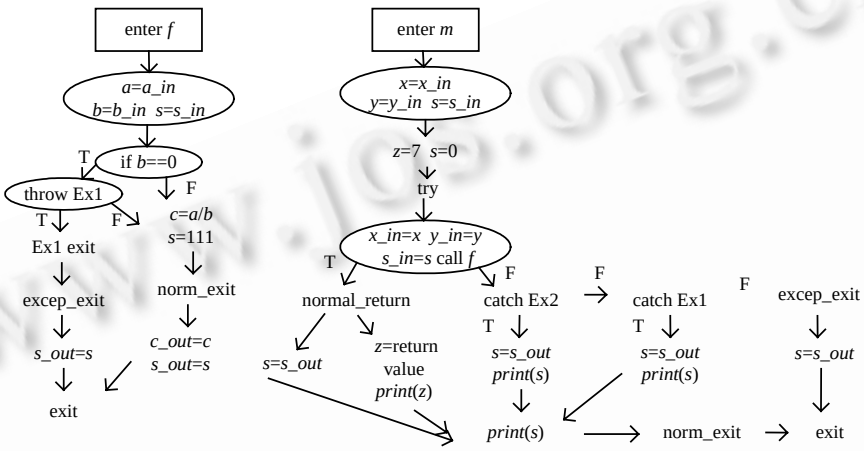

Fig.2 Improved control flow graph

图 2 改进控制流图 ICFG

3 函数间依赖性分析

为了分析在程序中出现的全局变量、函数调用、异常的传播等对程序依赖性的影响,必须对上一节建立的程序依赖图模型进行扩充.影响函数间依赖关系的主要因素有:

① 由于函数的调用而引起的控制依赖和数据依赖;

② 由于参数传递而引起的数据依赖;

③ 由于全局变量的作用而引起的数据依赖;

④ 由于异常的引发和异常的传播而引起的控制依赖和数据依赖.

3.1 函数间的依赖关系

函数调用的实质就是把控制转移到相应的函数.在转入函数之前,必须把实际参数传递给被调用的函数,即有一个形式参数与实际参数匹配的过程.

引发的异常如果在本函数内找不到匹配的处理程序,则异常会沿调用链逆向传播,传播到调用它的函数中继续寻找匹配的处理程序,这就引起了函数间的控制依赖和数据依赖.

具体来说,函数间的控制依赖边主要包括:

① 函数调用边和函数正常返回边;② 函数的异常返回边:如果函数调用语句在 try 块内,则函数的异常返回边从被调用函数的异常退出节点连到与调用函数的 try 块匹配的 catch 语句;如果函数调用语句不在 try 块内,则函数的异常返回边从被调用函数的异常退出节点连到调用函数的异常退出节点.

函数间的数据依赖关系主要包括:

① 实参与形参之间的依赖关系;② 全局变量和局部变量由于函数的调用、异常的传播所引发的数据依赖关系:函数间的数据依赖是与控制依赖密切相关的,函数调用点后的全局变量数据依赖于从被调用函数正常返回时的全局变量的值;catch 语句后的全局变量数据依赖于从被调用函数异常退出时的全局变量的值.

3.2 构建系统依赖图

扩充上一节的程序依赖图为系统依赖图,系统依赖图(SDG)[4]包含组成系统的各个函数的程序依赖图,用它来描述函数间数据依赖和控制依赖关系.构造算法见算法 1.该算法并不考虑被分析程序 P 中包含递归调用的情况,如果程序 P 中含有递归调用,需要参阅我们以前工作中对递归子程序中的依赖性分析的方法[3],再结合本文的方法进行分析,由于篇幅所限,我们将另文讨论.

算法 **1**. 系统依赖图的构造算法(algorithm for constructing SDG).

输入:PDG:在程序 P 中的每一个函数的 PDG;

输出:SDG:系统依赖图.

变量说明:functionlist:在程序 P 中调用的函数表.

begin ConstructSDG

1. 把程序 P 中的函数放入 functionlist;
2. **while** functionlist 非空
3. 从 functionlist 中移出函数 N(假设在程序 P 中函数 M 调用函数 N);
4. 增加数据依赖边从每一个 formal-in 节点到相应的 actual-in 节点;
5. 增加一条控制依赖边从函数 M 中调用函数 N 的调用点到函数 N 的入口节点;
6. 增加控制依赖边从函数 N 的 norm_exit 节点到函数 M 的调用节点的 normal_return 节点;
7. 增加数据依赖边从函数 N 的 norm_exit 的后继节点的 actual-out 节点到 normal_return 后继节点的 formal-out 节点;
8. **for** 函数 M 中且在 try 块内每一个调用函数 N 的调用点 **do**
9. **for** 函数 N 中的每个 excep_exit 节点 **do**
10. **if** 找到与 excep_exit 的异常类型相匹配的 catch 块 **then**
11. 增加控制依赖边从 excep_exit 节点到 M 中的相应 catch 节点;
12. 增加数据依赖边从 excep_exit 的后继节点的 actual-out 节点到 catch 节点的后继节点的 formal-out 节点;
13. **else while** 函数调用点在一个嵌套的 try 块中
14. 在上层的 catch 块中寻找与 excep_exit 中的异常类型相匹配的 catch 节点
15. **if** match **then**
16. 增加控制依赖边从 excep_exit 节点到该 catch 节点;
17. 增加数据依赖边从 excep_exit 的后继节点的 actual-out 节点到 catch 节点的后继节点的 formal-out 节点;
18. **endif**
19. **endwhile**
20. 增加控制依赖边从 excep_exit 节点到 M 的 excep_exit 节点;
21. 增加数据依赖边从 excep_exit 的后继节点的 actual-out 节点到 M 的 excep_exit 节点的后继节点的 formal-out 节点;
22. 把函数 M 添加到 functionlist;
23. **endif**
24. **endfor**
25. **endfor**
26. **for** 对函数 M 中且不在 try 块内每一个调用函数 N 的调用点 **do**
27. **for** 函数 N 中的每个 excep_exit 节点 **do**
28. 增加控制依赖边从 excep_exit 节点到 M 的 excep_exit 节点;

29. 增加数据依赖边从 excep_exit 的后继节点的 actual-out 节点到 *M* 的 excep_exit 节点的后继节点的 formal-out 节点;
30. 把函数 *M* 添加到 functionlist;
31. **endfor**
32. **endfor**
33. **endwhile**

end ConstructSDG

对每个被调用函数 *N*,增加函数调用边;从每一个 formal-in 节点到相应的 actual-in 节点增加数据依赖边;对每个正常退出节点(norm_exit 节点),增加控制依赖边和数据依赖边,这种连结方法与没有异常处理结构的程序中的函数调用的连结方式相同(4~7).对在函数 *M* 中且在 try 块内的函数 *N* 的调用点,增加控制依赖边从被调用函数 *N* 的 excep_exit 节点到相应的 catch 节点(如果找到匹配的处理程序),增加数据依赖边从 excep_exit 的后继节点的 actual-out 节点到 catch 节点的后继节点的 formal-out 节点(8~12).如果存在 try 块的嵌套且在本层的 catch 块中找不到匹配的处理程序,则到与上一层的 try 块关联的 catch 块中继续寻找合适的处理程序(13~19);如果最终还没有找到合适的处理程序,则联结 *N* 的 excep_exit 节点到调用函数的 excep_exit 节点作为控制依赖边,增加数据依赖边从 excep_exit 的后继节点的 actual-out 节点到 *M* 的 excep_exit 节点的后继节点的 formal-out 节点(20,21).对于函数 *M* 中且不在 try 块内每一个调用函数 *N* 的调用点,对函数 *N* 中的每个 excep_exit 节点,增加控制依赖边从 excep_exit 节点到 *M* 的 excep_exit 节点;增加数据依赖边从 excep_exit 的后继节点的 actual-out 节点到 *M* 的 excep_exit 节点的后继节点的 formal-out 节点(27~31).

图 3 是根据算法 1 得到的系统依赖图.在图中,实线表示控制依赖边,虚线表示数据依赖边;较细线表示函数内的依赖边,而较粗的线表示函数间的依赖边.

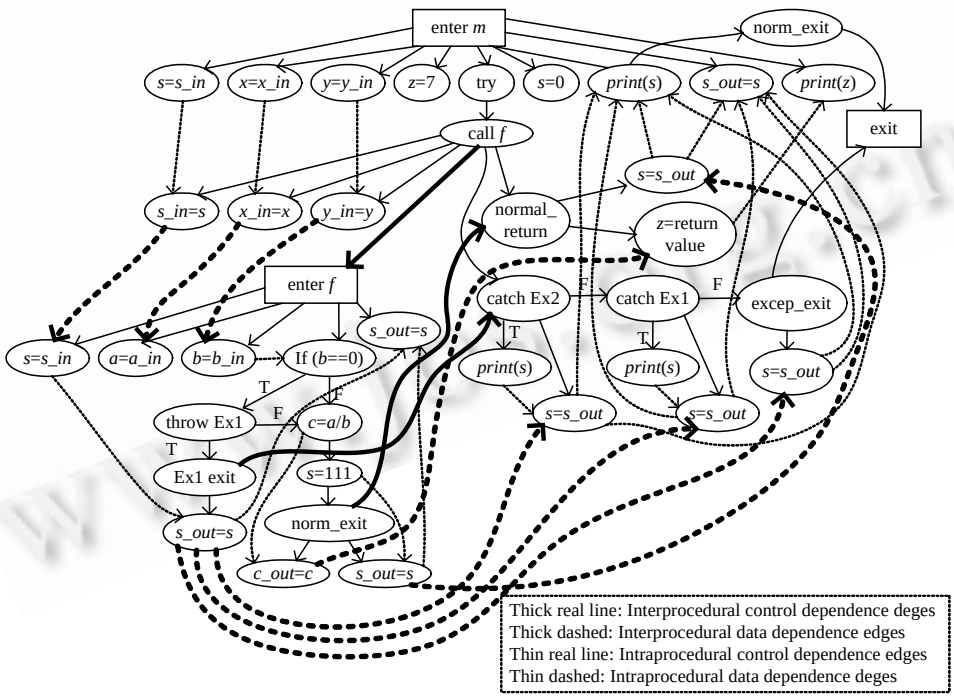

Fig.3 System dependence graph

图 3 系统依赖图

3.3 系统依赖图的特点

用上面的方法构建的系统依赖图与传统的系统依赖图相比,具有以下特点:

① 为了得到准确的控制依赖和数据依赖信息,我们把异常处理结构引入到语句层次的系统依赖图中.

② 由于 throw 语句和 catch 语句能够引起控制流的变化,所以把 throw 语句和 catch 语句也作为一个断言来处理,有“T”边和“F”边,这样更有利于信息的表示.

③ 对于函数的调用有两条返回边:一条是正常返回边,一条是异常返回边.当被调用函数在执行期间没有引发异常时,沿正常返回边返回,并返回到函数的调用点;当在被调用函数中引发异常时,将沿异常返回边返回,并且并不返回到调用点,而是在调用函数中寻找匹配的处理程序.这在系统依赖图中能够很清晰地表示出来.

④ 从系统依赖图中也可以很清楚地表示出某个异常的传播路径.只要在系统依赖图上从异常的引发点开始,沿着控制依赖边走到它的处理程序为止,即是该异常的传播路径.

⑤ 由于在系统依赖图中我们把全局变量作为函数的参数来处理,所以在系统依赖图中可以很好地表示出全局变量的数据依赖关系.

⑥ 我们的系统依赖图只表示出了显式引发的异常对程序的依赖性分析的影响,对于隐式引发的异常并没有表示出来.如果把程序中所有可能隐式引发异常的语句都用断言来表示,则系统依赖图将会相当复杂.但完全不考虑隐式引发异常所带来的影响也是不合适的,如何表示和适度地考虑隐式引发异常对程序依赖性的影响将是我们下一步研究的主要工作.

4 应用分析

程序切片的概念最早由 Weiser 提出[4],对程序 P、程序点 s 和变量 X,其切片 $SLICE(X,s,P)$由程序 P 中的部分语句和谓词组成,这些语句和谓词的执行可能影响 s 处变量 X 的值,其中,$\langle s,X\rangle$为一切片标准.

一般情况下,切片分析用于计算程序中关于某个语句 s 的切片,即切片标准为$\langle s\rangle$,它等价于$\langle s,Ref(s)\cup Def(s)\rangle$,其中,$Ref(s)$表示对语句 s 中的所有元素的引用,$Def(s)$表示对语句 s 中的所有元素的定值.由于顺序程序中依赖关系具有传递性,基于程序依赖图的函数内切片算法是一个简单的图的可达性问题.在系统依赖图上,函数间的切片问题也是一个图的可达性问题.Horwitz 等人介绍的切片算法共有两遍,最后的程序切片是这两步所标记节点的并集[4].两遍遍历过程如下所示:

① 在系统依赖图上,后向遍历除 *parameter-out* 边外的所有边,并且标记可达节点;

② 从上一步标记的节点开始,后向遍历除 *call* 和 *parameter-in* 边的所有边,并标记可达节点.

把这种方法应用到我们构建成的系统依赖图上,就可以得到包含异常处理结构的程序的切片.如图 1 所示,第 21 行的输出语句的关于 s 的切片是图中的语句前带“+”符号的语句;如果不考虑 throw 和 catch 的特殊性,只是作为一般语句来处理,则第 21 行的输出语句的关于 s 的切片是图中的语句前带“*”符号的语句,从这个例子来看,精度提高了将近 31%.因此,通过异常分析带来了切片精度的提高.

5 相关工作比较

目前,针对程序依赖性分析的文章大多没有考虑异常处理结构对其造成的影响.在进行异常分析的过程中,同时进行依赖性分析的文章也较少.相关的研究工作主要有下面几种:

Choi 等人在 1999 年提出的一种用要素控制流图(factored control-flow graph)表示程序中的过程内的控制流的方法[6].这种表示方法简化了传统的控制流图方法,它既可以表示显式引发的异常所引起的控制流变化情况,也可以表示隐式引发的异常所引起的控制流变化情况.但该方法仅限于数据流的分析,并没有考虑异常处理结构对程序依赖性分析的影响.

Shelekhov[12]等人在进行 Java 程序的数据流分析时,对引发异常的隐式控制流显式地、动态地构造出来.在我们的改进控制流图中,也把引发异常的隐式控制流显式地表示出来,如 throw 节点有“T”和“F”两条出边等.但他们并没有进行依赖性的分析.

Sinha[5,7,8]等人对支持异常处理结构的 Java 程序作过依赖性分析和切片分析,其分析方法与我们的分析方法有相似之处,但这种表示方法当从 try 到 throw 之间的调用链超过 1 时,过程间的依赖性分析不能准确地表示[9],而且也没有考虑到异常处理结构对数据依赖性的影响,只讨论了异常处理结构对控制依赖的影响.

Allen[9]等人在对 Java 程序进行切片分析时,考虑了异常处理结构对程序切片的影响,但存在两个问题:一是 throw 语句的 true 边并不一定会连到“exceptional exit”节点,只有 throw 语句不在 try 块内,才会首先连到“exceptional exit”节点;如果 throw 语句在 try 块内时,应该首先连到 catch 节点;二是 catch 语句的 false 边也不一定连到紧跟在最后一个 catch 块后第一个语句,因为如果该 catch 语句与引发的异常类型不匹配,并且没有 finally 块,则程序不会执行紧跟在最后一个 catch 块后的第一个语句,而是退出该 try 层或退出该方法,所以应该连到与 try 匹配的这组 catch 块中的下一个 catch 语句(如果有的话),或上一层的 catch 语句(如果有 try 块嵌套),或连到该函数的“exceptional exit”节点.我们的表示方法很好地解决了以上这些问题.

Robillard [13]等人描述了一个分析 Java 程序中捕获不到的异常模型,并提供了一个静态分析工具 Jex.但该模型并没有对程序的依赖性进行分析,只是从异常变量的异常类型上对可能引发异常的捕获情况进行了分析.

Fu[14]等人在对网络服务应用的健壮性进行测试时,对 JDK 库所抛出的隐式检查型异常的数据流进行了分析,但他们并没有对程序的依赖性进行分析.

6 结束语

针对 C++的异常处理机制,本文改进了传统的控制流图,使其能够描述异常处理结构,在此基础上建立了一种函数内和函数间的依赖图模型.在以前工作的基础上,扩展传统的依赖性分析方法来适应含有异常处理结构的 C++程序的依赖性分析,并讨论了这种依赖性分析方法在程序切片中的应用.我们曾在软件度量方面做过比较深入的研究,也取得了一定的成果[10,15,16],异常的传播增加了函数之间的耦合度,给程序的理解和分析带来了一定的困难,在此依赖性分析的基础上,我们下一步将对异常传播对软件度量的影响做更深入的研究.

References:

[1] Chen ZQ, Xu BW, Liu KC, Yang HJ, Liu KC, Zhang JP. An approach to analyzing dependency of concurrent programs. In: Proc. of the IEEE APAQS 2000. Hong Kong, 2000. 39−43. http://doi.ieeecomputersociety.org/10.1109/APAQ.2000.883776

[2] Chen ZQ, Xu BW. Dependency analysis based dynamic slicing for debugging. Wuhan University Journal of Natural Sciences, 2001, 6(1-2):398−404.

[3] Xu BW, Zhang T, Chen ZQ. Dependence analysis of recursive subprograms and its applications. Chinese Journal of Computers, 2001,24(11):1178−1184 (in Chinese with English abstract).

[4] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. ACM Trans. on Programming Languages and Systems, 1990,12(1):26−60.

[5] Sinha S, Harrold MJ. Analysis and testing of programs with exception-handling constructs. IEEE Trans. on Software Engineering, 2000,26(9):849−871.

[6] Choi JD, Grove D, Hind M, Sarkar V. Efficient and precise modeling of exceptions for the analysis of Java programs. In: Proc. of the ’99 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 1999. 21−31. http://www.research.ibm.com/jalapeno/pub/paste99.ps

[7] Sinha S, Harrold MJ. Analysis of programs with exception-handling constructs. In: Proc. of the ICSM. 1998. 348−357. http://ieeexplore.ieee.org/iel4/5960/15947/00738526.pdf

[8] Sinha S, Harrold MJ, Rothermel G. System-Dependence-Graph-Based slicing of programs with arbitrary interprocedural control flow. In: Proc. of the Int’l Conf. on Software Engineering. 1999. 432−441. http://citeseer.ist.psu.edu/5475.html

[9] Allen M, Horwitz S. Slicing Java programs that throw and catch exceptions. In: Proc. of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. 2003. 44−54. http://www.cs.wisc.edu/wpis/papers/pepm03.ps

[10] Chen ZQ, Zhou YM, Xu BW, Zhao JJ, Yang HJ. A novel approach to measuring class cohesion based on dependence analysis. In: Proc. of the Int'l Conf. on Software Maintenance (ICSM 2002). 2002. 377−383. http://ieeexplore.ieee.org/iel5/8357/26332/01167794.pdf

[11] Chen ZQ. Program slicing based on dependence analysis [Ph.D. Thesis]. Nanjing: Southeast University, 2003 (in Chinese with English abstract).

[12] Shelekhov VI, Kuksenko SV. Data flow analysis of Java programs in the presence of Exceptions. In: Broy BM, Zamulin A. eds. Proc. of the Perspectives of System Informatics, the 3rd Int'l Andrei Ershov Memorial Conf. (PSI'99). LNCS 1755, Springer-Verlag, 2000. 389−395. http://citeseer.ist.psu.edu/shelekhov99data.html

[13] Robillard MP, Murphy GC. Static analysis to support the evolution of exception structure in object-oriented systems. ACM Trans. on Software Engineering and Methodology, 2003,12(2):191−221.

[14] Fu C, Ryder BG, Wonnacott DG. Robustness testing of Java server applications. IEEE Trans. on Software Engineering, 2005,31(4):292−311.

[15] Chen ZQ, Xu BW. An approach to measurement of class cohesion based on dependence analysis. Journal of Software, 2003,14(11):1849−1856 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/1849.htm

[16] Chen ZQ, Xu BW, Zhou YM. Measuring class cohesion based on dependence analysis. Journal of Computer Science and Technology, 2004,19(6):859−866 (in English with Chinese abstract).

附中文参考文献:

[3] 徐宝文,张挺,陈振强.递归子程序的依赖性分析及其应用.计算机学报,2001,24(11):1178−1184.

[11] 陈振强.基于依赖性分析的程序切片技术研究[博士学位论文].南京:东南大学,2003.

[15] 陈振强,徐宝文.一种基于依赖性分析的类内聚度度量方法.软件学报,2003,14(11):1849−1856. http://www.jos.org.cn/1000-9825/14/1849.htm

姜淑娟(1966－),女,山东莱阳人,在职博士生,副教授,主要研究领域为程序设计语言,软件分析与测试,编译技术.

徐宝文(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为程序设计语言及其理论与实现技术,软件开发方法与技术,软件分析测试与质量保证,软件逆向工程与软件再工程,知识与信息获取技术,基于 Web 系统及其分析测试技术.

史亮(1979－),男,博士生,主要研究领域为程序分析与测试.

周晓宇(1972－),男,在职博士生,讲师,主要研究领域为程序设计语言,软件工程.