

一种包含异常处理结构的面向对象切片方法

郝 杰^{1,2}, 姜淑娟²

(1 徐州医学院 计算机教研室, 江苏 徐州 221004; 2 中国矿业大学 计算机科学与技术学院, 江苏 徐州 221116)

摘 要: 针对面向对象语言的特点, 结合异常处理机制, 对传统的系统依赖图(SDG 图)进行了扩展, 实现了面向对象的系统依赖图(OSDG 图). 改进后的 OSDG 图可以很好地支持继承和多态等特征, 并能处理多态对象抛出和处理异常所带来的数据和控制依赖关系. 通过传统的程序切片算法, 可以在 OSDG 上切出较为精确的切片.

关键词: 面向对象; 异常处理; 程序切片; 系统依赖图; 控制流图

中图分类号: TP311

文献标识码: A

文章编号: 1000-7180(2012)03-0001-05

An Approach of Slicing for Object-Oriented Language with Exception Handling

HAO Jie^{1,2}, JIANG Shu-juan²

(1 Computer Department, Xuzhou Medical College, Xuzhou 221004, China; 2 School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China)

Abstract: Based on analyzing the object-oriented language feature and exception handling mechanism, this paper make some improvements to traditional system dependence graph (SDG), realize Object-oriented system dependent graph (OSDG). The improved OSDG can clearly describe some complex problems for object-oriented languages such as inheritance, polymorphism and deal with data and control dependence for the exception thrown out, and exception treatment from polymorphism object. Through the traditional slicing algorithm, more accurately slice can be gotten.

Key words: object-oriented; exception handing; program slicing; system-dependence graph; control-flow graph

1 引言

程序切片主要用于程序理解、程序分析、逆向工程、测试、调试等多个方面, 因此对程序切片的研究有着极其重要的意义. 面向过程语言的切片技术已趋于成熟.

然而, 对于面向对象语言的特点, 如继承, 多态以及动态联接, 使原来的分析技术不能适用于面向对象的程序. 随后研究者们提出了面向对象的程序切片方法, 郝学皎^[1]等人针对并发程序进行了面向对象程序切片方法的改进, 但是对于异常传播对面向对象程序切片的影响却疏于考虑. 异常结构是对程序运行中可能出现的错误进行处理, 这在现代程序设计和编写中是广泛存在的. 虽然 Sinha^[2]提出

了异常结构的表示方式, 姜淑娟^[3]等人对此进行了改进, 但他们的研究都限于单个类内部方法间的异常传播, 这不能满足当前面向对象软件开发的实际需要.

因此, 本文在 D Liang^[4]提出的对象切片的基础上构造了一种包含异常处理结构的系统依赖图, 这种方法可以处理多态对象的异常传播. 然后, 通过一个实例演示了如何构建 OSDG 以及如何用它计算切片.

2 异常处理结构对面向对象程序依赖性的影响分析

在运行程序中引发一个异常时, 异常处理机制会使程序的正常的控制流转变到异常的控制流, 从

收稿日期: 2011-07-21; 修回日期: 2011-08-15

基金项目: 国家自然科学基金(60970032); 教育部科学技术研究重点项目(108063); 江苏省自然科学基金(BK2008124)

而引起控制依赖的变化. 当程序的控制流发生变化必然引起程序中某些变量的定值——引用链的变化, 因此又会使数据的依赖性发生改变. 根据姜淑娟^[3]提出的改造包含异常处理结构控制流图(CFG)的方法, 对图 1 所示程序基类中的 vm() 和 m1() 控制流图进行改造, 改造后的 CFG 如图 2 所示.

3 面向对象过程依赖图(OPDG)的构造方法

在面向对象的程序中, 由于类间存在的关系可能会引起它类中某些方法的执行, 因此不能只在当前类下构造程序依赖图(PDG). 利用类调用图来完成判断.

首先, 标记类中声明的方法, 然后, 如果此类继承了某个基类, 那么标记在基类中的方法. 通过类调用图从标记过的方法开始进行反向遍历, 对能够到达的方法进行标记, 所有标记过的方法都需要新的 PDG. 文中把改进后的 PDG 称之为面向对象的程序依赖图(OPDG).

改进方法如下:

<pre> 1 Class Derived ;Public Base { 2 long d; * 3 vm(int e){ * 4 b=1 * 5 if (d>0) * 6 throw new E1(); * 7 b=2+e; } 8 public: 9 Derived():Base(){ * 10 d=0; } 11 m3(){ 12 m2(1) } };//end of class </pre>	<pre> * 25 m1(int x){ * 26 try{ * 27 vm(x) * 28 cout<<b<<endl; } * 29 catch(E1 &e1){ * 30 cout<<b<<endl; } 31 catch(E2 &e2){ 32 cout<<b<<endl; } * 33 cout<<b<<endl; //end of m1 };//end of class </pre>
<pre> 13 Class Base { 14 int b; 15 virtual vm(int e){ 16 b=e 17 if(b>0) 18 throw new E2(); 19 b=e+1 } 20 public: 21 Base(){ 22 b=0; } 23 m2(int i){ 24 b=b+i; }//end of m2 </pre>	<pre> * 34 C(Base &ba){ * 35 ba.m1(2); } * 36 main(){ 37 int i; 38 Base * p; * 39 cin>>i; * 40 if (i>0) 41 p=new Base (); * 42 else * 43 p=new Derived(); * 44 C(*p); } </pre>

图 1 程序示例

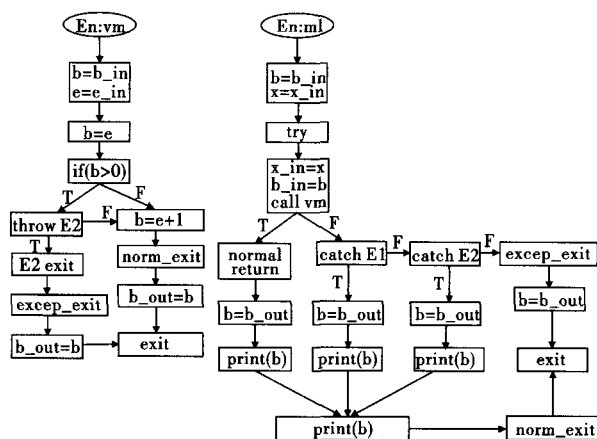


图 2 基类 vm() 和 m1() 的 CFG

(1) 处理参数对象

如果对象被当作一个参数, 就把参数对象表示成一棵树: 参数对象表示成树根, 参数对象的数据成员表示成叶子, 用边来表示对象和数据成员的数据依赖. 如果参数对象的数据成员是另外的一个对象, 那么接着进行树的扩展, 形成子树.

如图 3 所示, 仅表示基类 C(Base &) 函数的 OPDG. 形式参数 ba 被表示成一棵树, 树的叶子结点表示对象的数据成员 b. 数据流边连接着对象和数据成员结点.

(2) 处理多态对象

在 OPDG 中多态对象也被表示成一棵树: 树根表示多态对象结点, 孩子结点表示不同类型的对象. 如果多态对象作为了一个参数, 那么将其扩展成一棵子树. 当多态对象接收到一个消息, 就把孩子结点扩展成一个调用结点, 对于每一个可能的对象类型, 都用不同的调用结点来表示, 不同的调用结点执行不同的虚函数.

如图 4 表示在执行完 main() 函数后(图 1 所示程序)C(Base &) 的 OPDG.

(3) 处理异常

以上述分析为基础, 增加相应的依赖边, 即可完成带有异常结构的 OPDG 的构造. 见图 5 中相应的代表处理异常的 m1(x) 方法的 OPDG 的表示. 由于在基类声明的方法 m1() 调用了虚函数 vm(), 在基类和继承类中 vm() 分别抛出两个类型的异常. 因此, 要创建两个不同的调用 m1() 的 enter 结点, 用不同类型的 enter 结点(“Base”, “Derived”)分别连接不同的 formal-in/out 结点和虚函数调用结点等, 这样的表示能使处理异常函数的数据依赖和控制依赖更加精确.

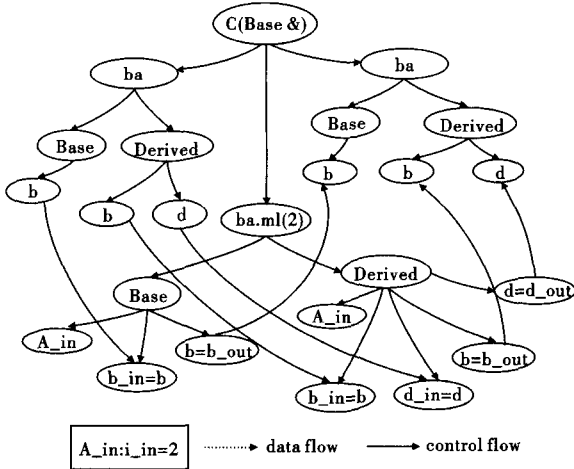


图3 基类中 C(Base &)的 OPDG

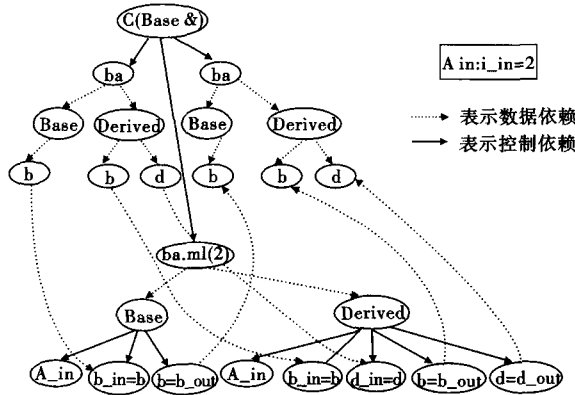


图4 执行完 main() 函数后, C(Base &)的 OPDG

类似地,用两个 enter 结点(“Base”,“Derived”)分别表示进入两种不同类型的虚函数,它们抛出了不同类型的异常。见图5中相应的代表抛出异常方法 vm(e)的 OPDG 的表示。

4 面向对象系统依赖图(OSDG)的构造算法

将上一节改进的 OPDG 扩展为面向对象的系统依赖图(OSDG)。构造 OSDG 的算法如算法1所示。

算法1 对象系统依赖图(OSDG)的构造算法 (algorithm for constructing OSDG).

输入:OPDG;在程序中每个函数的 OPDG;

输出:OSDG;对象系统依赖图。

变量说明: functionlist;程序调用的函数列表。

begin ConstructOSDG

1. 将程序中的所有函数放入 functionlist;

2. while functionlist 非空

3. 把函数 Q 移出 functionlist(如果函数 P 调用函数 Q)

4. if Q 函数的参数是对象 then

5. 连接 formal-in 到相应的 actual-in 对象参数结点作为数据依赖边;

6. 连接函数 P 调用 Q 的调用点到函数 Q 的入口结点作为控制依赖边;

7. if Q 函数的对象参数为多态对象 then

8. 连接 P 函数各种类型的对象参数结点和 Q 函数相应类型的对象参数结点作为数据依赖边;

9. end if

10. else if Q 是虚函数 OR Q 调用了虚函数 then

11. 去除在 Q 函数 OPDG 中的“Base”“Derived”等表示不同类型的结点;

12. 分别连接在基类和继承类中相应的 formal-in 到 actual-in 参数结点作为数据依赖边;

13. 分别连接函数 P 的调用点到函数 Q 的入口结点作为控制依赖边;

14. else

15. 连接从每个 formal-in 结点到相应 actual-in 结点作为数据依赖边;

16. 连接函数 P 调用 Q 的调用点到函数 Q 的入口结点作为控制依赖边;

17. end if

18. end if

19. if P 函数的 OPDG 中含有 normal_return 结点和 Q 函数的 OPDG 中含有 norm_exit 结点 then

20. 连接 Q 的 norm_exit 结点到 P 的 normal_return 结点作为控制依赖边;

21. 连接 Q 的 norm_exit 结点的后继结点 actual-out 结点到 normal_return 后继结点 formal-out 结点作为数据依赖边;

22. else

23. 连接 Q 的 actual-out 结点到 P 的 formal-out 结点作为数据依赖边;

24. end if

25. for 函数 P 中且在 try 块内每一个调用 Q 的调用点 do

26. for 函数 Q 中每个 excep_exit 结点 do

27. if 找到与 excep_exit 异常类型相匹配的 catch 块 then

28. 连接从 excep_exit 结点到 P 中的相应 catch 结点作为控制依赖边;

29. 连接从 excep_exit 的后继结点 actual-out 结点到相应的 catch 结点的后继结点 formal-out 结点作为数据依赖边;

30. else while 函数调用点在一个嵌套的 try 块中

31. 在上层的 catch 块中寻找与 excep_exit 中的异常类型相匹配的 catch 结点;

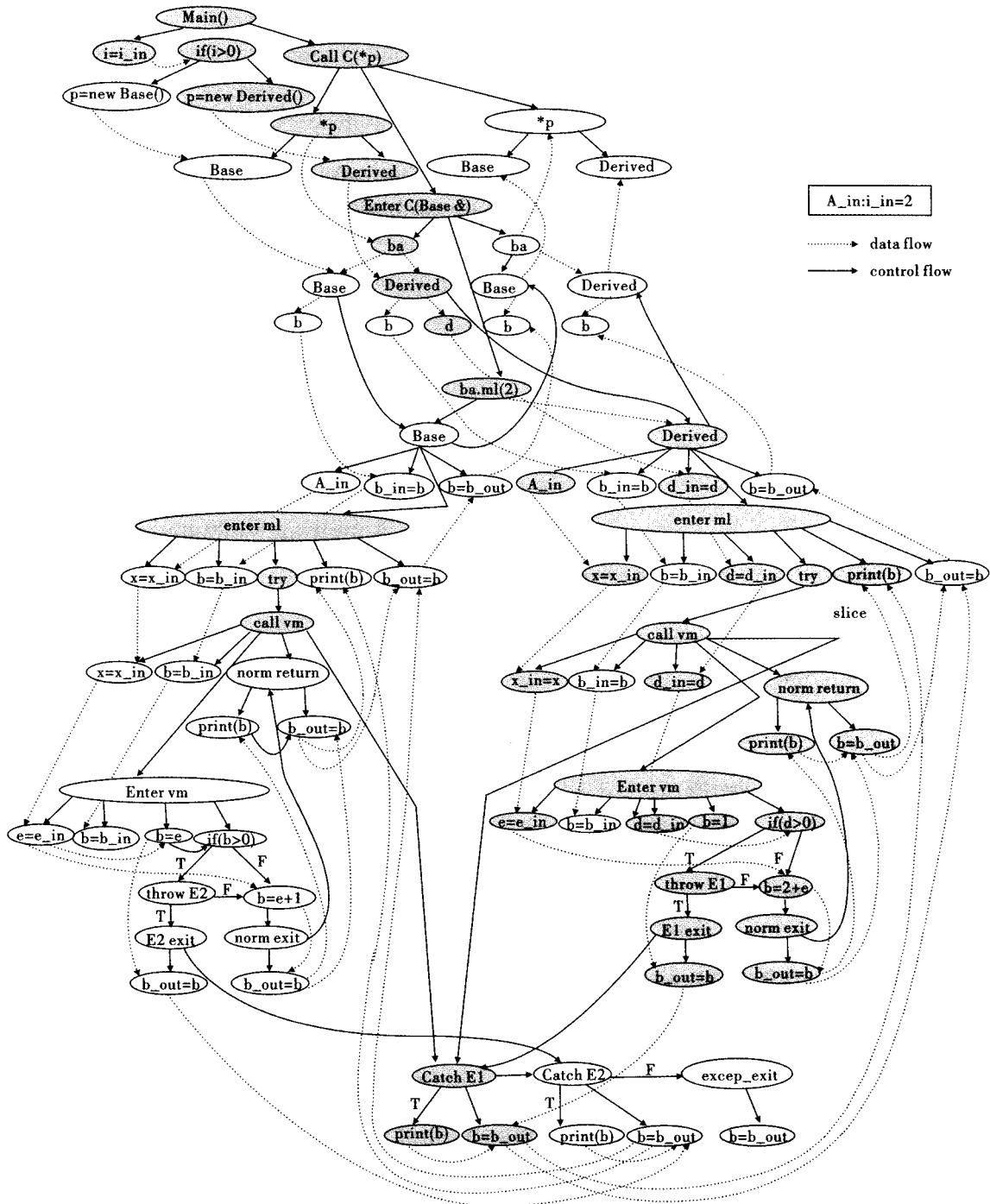


图 5 程序的 OSDG

32. if 匹配 then

33. 连接从 excep_exit 结点到该 catch 结点作为控制依赖边;

34. 连接从 excep_exit 的后继结点 actual-out 结点到相应的 catch 结点的后继结点 formal-out 结点作为数据依赖边;

35. end if

36. end while

37. 连接从 excep_exit 结点到 P 的 excep_exit 结点作为

控制依赖边;

38. 连接从 excep_exit 后继结点 actual-out 结点到 P 的 excep_exit 后继结点 formal-out 结点作为数据依赖边;

39. 把函数 P 添加到 functionlist 中;

40. end if

41. end for

42. end for

43. end while

44. end constructOSDG

表1 不同方法的切片结果比较

方法比较	处理异常范围	Derived::m1()方法中第33行关于b的切片结果	切片是否精确
姜淑娟[3]	类内方法间	15,16,17,18,19,25,26,27,28,29,30,31,32,33	×
D Liang [4]	不能处理异常	3,4,5,7,10,25,33,34,35,36,39,40,42,43,44	×
张艳梅[5]	类间	15,16,17,18,19,25,26,27,28,29,30,31,32,33,34,35,36,39,40,42,43,44	×
本文	类间	3,4,5,6,7,10,25,26,27,28,29,30,33,34,35,36,39,40,42,43,44	✓

图5是根据算法1得到的面向对象的系统依赖图.通过图5可以清楚地分析出在基类和继承类中的两个虚函数分别抛出的不同异常与调用虚函数m1()中处理异常的各种依赖关系: Catch E1 结点同时控制依赖于E1_exit和E2_exit结点,如果类型匹配不上,则继续向后寻找匹配. Catch E1 结点的后继b=b_out数据依赖于Derived::vm()中E1_exit结点的后继b_out=b,而Catch E2结点的后继b=b_out数据依赖于Base::vm()中E2_exit结点的后继b_out=b.不仅如此,在图5中还能清楚地表示出在运行过程中由于调用虚函数,主调函数里的变量与被调用虚函数里的变量的数据依赖关系:如果m1函数在运行过程中调用了Base::vm()/Derived::vm(),则enter m1()的后继b_out=b分别数据依赖于在Base/Derived类中call vm结点下的norm_exit的后继b=b_out(如果在Base::vm()/Derived::vm()中没有抛出异常)和Catch E2/Catch E1结点的后继b=b_out(如果在Base::vm()/Derived::vm()中抛出了异常).

5 切片方法

在构造出OSDG后,我们用传统两步程序切片算法得到切片.把Derived::m1()方法中第33行关于b的切片在图5中用阴影标出,在图1中用“*”标出.将文献[3-5]的方法和本文方法得出的切片结果进行比较,如表1所示.

从表1可以看出,文献[3]只能处理类内部方法间的异常,文献[4]只对普通类间切片进行了分析,无法处理带有异常传播的特殊情况,文献[5]虽然可以分析带有异常传播的类间切片但是无法处理多态对象及多态对象做参数等复杂情况,因此以上切片结果都不准确.而本文提出的方法不但可以处理类间的异常而且可以在OSDG中准确分析出多态对象做参数时的切片结果.当 $i \leq 0$ 时,多态对象ba属于Derived类,因此执行函数m1()时调用Derived类中的虚函数vm()来抛出异常,这样计算出的切片都属于Derived类而与Base类没关系.

6 结束语

异常结构在现代程序设计和编写中广泛存在,但目前构造SDG的方法大都局限于类内部方法间的异常传播,不能有效地解决面向对象语言的特征.因此本文提出了一种面向对象的异常传播表示方法,并给出OSDG图的构造算法.利用OSDG可以表示出在不同虚函数下抛出异常和在调用不同虚函数时处理异常的问题,同时还可应用于包含异常处理结构的面向对象程序切片中.

由于目前的OSDG图比较复杂,如何进行简化是下一步工作的方向.

参考文献:

- [1] 郝学蛟,汤小春. 基于依赖标识的并发程序动态切片方法[J]. 微电子学与计算机, 2009, 26(7): 5.
- [2] Sinha S, Harrold M J. Analysis of programs with exception handling constructs[C] // Proceedings of the International Conference on Software Maintenance. Washington, 1998:348-357.
- [3] 姜淑娟,徐宝文,史亮,等. 一种基于异常传播分析的依赖性分析方法[J]. 软件学报, 2007, 18(4): 10.
- [4] Liang D. Slicing objects using system dependence graphs[C] // Int'l Conf on Software Maintenance. USA: Maryland, 1998:358-367.
- [5] 张艳梅,姜淑娟,袁冠. 一种包含异常传播的类间数据依赖分析方法[J]. 软件时空, 2010, 26(2-3): 201-203.

作者简介:



郝杰 女,(1980—),讲师,硕士研究生.研究方向为程序设计语言、异常处理等.

姜淑娟 博士,教授,博士生导师.