一个分层切片工具模型*)

A Hierarchical Slicing Tool Model

谭 毅 朱 平 李必信 郑国梁

(南京大学计算机软件新技术国家重点实验室 南京大学计算机系 南京210093)

Abstract Most of the traditional methods of slicing are based on dependence graph. But constructing dependence graph for object oriented programs directly is very complicated. The design and implementation of a hierarchical slicing tool model are described. By constructing the package level dependence graph, class level dependence graph, method level dependence graph and statement level dependence graph, package level slice, class level slice, method level slice and program slice are obtained step by step.

Keywords Program slice, Hierachical slice, Dependence graph, Two passes algorithm

程序切片技术在程序调试,测试、程序理解、逆向工程和软件维护等方面有着广泛的应用,程序切片[5]是一组可能影响到在程序中某个点 i 的某个变量 v 的值的语句或谓词的集合。而(v,i)被称作切片准则。这里 v 也可以是一组变量。自从 Mark Weiser 提出切片的概念,随着程序依赖图[6]、系统依赖图[5]的出现,传统程序的切片技术已走向成熟。1994年以来,面向对象的程序切片逐渐成为研究的主流。A. Krishnaswamy 利用一种面向对象的程序依赖图(OPDG,Object-Oriented Program Dependence Graph)[1]来计算面向对象程序的语句切片,但是 OPDG 不能表示动态绑定等问题。D. Liang, L. D. Larsen 和 M. J. Harrold 扩展系统依赖图来计算面向对象程序的切片,在一定程度上解决了动态绑定和对象参数的问题[3,4]。这些切片方法都是基于依赖图的、而构造 OO 程序的依赖图是一件非常复杂的工作、而且构造过程中容易出错,这会导致切片的结果不正确,造成前功尽弃。李必信[8]提出了分层切片的思想,利用逐步求精的方法来得到面向对象程序的切片。

本文首先描述 Java 分层切片工具模型 JSTM(Java Slicing Tool Model)的设计思想,然后描述了 JSTM 的实现,并对该模型中的切片算法与以往的一些方法进行比较。最后是结论。

1 JSTM 的设计思想

为了达到软件理解、逆向工程的目的,有时不必切片到程序的每一个语句,只需切片到方法或类就够了。而面向对象的程序有着清晰的层次结构,Java 语言层次结构更为简单:包由类和接口组成,类和接口由方法和成员变量组成,方法由语句组成。JSTM 在这几个层次上分别构造不同的依赖图,并得到切片,以满足不同的需要。

为了便于表述,我们提出下面一组概念:

包级切片:程序关于切片准则(v,i)的包级切片是该类中可能影响到程序中点 i 的某个变量 v 的值的包的集合,这里提到的包也就是 Java 语言中的 Package。

类级切片:程序关于切片准则(v,i)的类级切片是程序中可能影响到程序中点 i 的某个变量 v 的值的类和接口的集合。

方法级切片:程序关于切片准则(v,i)的方法级切片是程序中可能影响到程序中点 i 的某个变量 v 的值的类或接口的方法和成员变量的集合。

需要指出,只要某个包、类、接口或方法中包含一条语句或谓词可能影响某个变量 v,我们就说这个包、类,接口或方法可能影响 v。如果存在一条语句或谓词可能影响变量 v,而这条语句又使用了变量 i,我们就说变量 i 可能影响变量 v。

分层切片模型把 Java 源程序按逻辑结构分为四个层次,即包层、类或接口层、类或接口中的成员方法和成员数据层、方法中的语句和变量层。Java 源文件由若干个包组成,这些包之间通过 import 语句而发生联系;一个包中有若干个类或接口声明,这些类和接口之间通过继承、实现、类成员的调用而形成复杂的依赖关系;一个类或接口声明中包含若干个成员数据和成

*)本文工作得到江苏省自然科学基金(编号 BK99038)和江苏省应用基础研究(编号 BJ2000034)资助。

员方法，方法之间通过互相调用而发生联系；方法则由若干条语句和若干个变量（包含局部变量和全局变量）实现。

基于分层切片模型和切片准则，得到如下面向对象的分层切片算法的主要思想：考虑切片准则(v,t)，其中 v 和 s 分别是任何一个方法中的一个变量或变量集合和一条语句。使用分层切片方法来计算关于切片准则(v,i)的程序切片（下面的 P 代表可能包括多个包的 java 源程序）：

①包层：首先确定所有与包含 v 和 s 的类有关（由 import 语句而形成的直接或间接依赖关系）的包，删除那些与包含 v 和 s 的类无任何关系的包。通过这种方法得到关于切片准则(v,i)的包级切片 S'(P)。

②类或接口层：对 S'(P)中的类层次进行逐步求精，进一步计算该类层次中影响切片准则的类，并把不影响切片准则或不受切片准则影响的类从类层次结构中删除，从而得到关于切片准则(v,i)的类间切片 S''(P)，并且满足 S'(P)⊇S''(P)。

③成员方法和成员数据层：在上层基础上进一步计算单个类中影响切片准则或受切片准则影响的成员方法和成员数据，删除那些与切片准则无关的成员方法和成员数据，然后得到类内切片 S'''(P)，且有 S'(P)⊇S''(P)⊃S'''(P)。

④语句和局部变量层：在上面工作的基础上进一步计算方法中与切片准则有关的变量和语句，删除那些与切片准则无关的变量，语句和控制谓词等，这一步的工作与传统的面向过程的程序切片一样，最后得到程序切片 S(P)，同时满足 S'(P)⊇S''(P)⊃S'''(P)⊇S(P)。到此，我们得到一个面向对象程序关于切片准则(v,t)的程序切片 S(P)。

2 JSTM 的实现

在 JSTM 实现时，我们确定 Java 的一个语法子集，而不考虑多线程、异常等问题。JSTM 可分为三层：语法分析层、依赖图生成层、切片层。如图1所示。

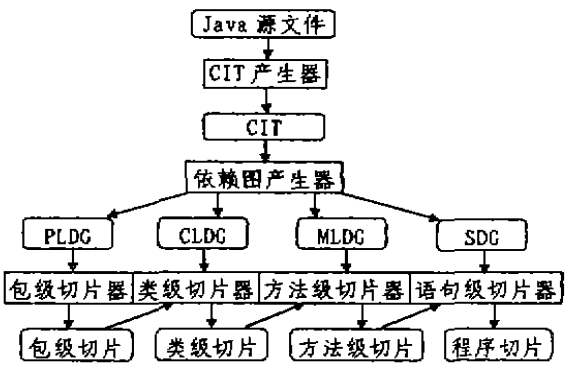

图1 层次切片工具的体系结构

语法分析层对 Java 源程序进行传统的语法语义分析，并对得到的信息进行加工得到代码信息树（CIT，Code Information Tree）和符号表。在实践中，我们用 lex 和 yacc 得到这一层模块。具体算法在本文中不再详述。

依赖图生成层利用 CIT 中的信息，生成包级依赖图（PLDG）、类级依赖图（CLDG）、方法级依赖图（MLDG）和语句级依赖图。

切片生成层利用四种依赖图逐级生成包级切片、类级切片、方法级切片和程序切片。

2.1 依赖图生成层

2.1.1 包级依赖图（Package Level Dependence Graph，PLDG） 用来表示包与包之间的引入和子包关系，PLDG 是一个二元组(Np,E)，Np 是包的集合，E 是边的集合。每一个结点代表一个包，如果包 p1 中引入了包 p2，则从结点 p2到结点 p1有一条边。

由于 Java 语言中包的引入关系是通过 import 语句建立的，包级依赖图可以通过扫描 CIT 直接得到。

```
1  Class A{
2  public int x;
3  public f1(){
4  x=1;}
   }
5  class B extends A{
6  public int y;
6  public f1(){
7  x=2;}
8  public f2(){
9  y=3;}
   }
10  class C extends A{
11  public int y;
12  public f1(){
13  x=y;}
    }
21  class D{
22  public void f3(A a){
23  a.f1();
24  return;
     }
    }
25  Class E{
26  public static void main(String str[]){
27  A a1=new B();
28  D d1=new D();
29  d1.E(a1);
 }
}
```

图2 程序1

2.1.2 类级依赖图（Class Level Dependence Graph，CLDG） 用来表示类或接口之间的继承、实现和创建关系，PLDG 也是一个二元组(Nc,E)，Nc 是类的集合，E 是边的集合。若类 C1 中直接创建了一个属于类 C2的对象，在这里不妨说 C1和 C2有创建关系。如果类 A 中直接创建了类 B 的对象，则 AB 之间有一条双向的"创建边"。如果类 B 继承了类 A 或实现了接口 A，则有一条继承边从 B 到 A。另外假设所有的静态函

数和静态变量，它们属于一个不存在的类 Static。

由于 Java 中类或接口的继承、实现和创建关系分别由 extend、implement 和 new 说明，GLDG 也可以通过扫描 CIT 直接得到。图2中的程序1的 CLDG 如图3所示。

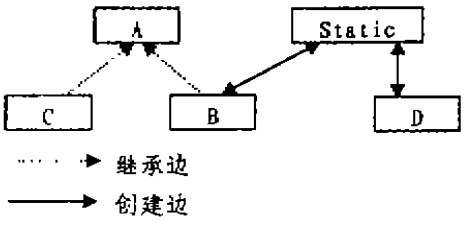

图3 程序1的 CLDG

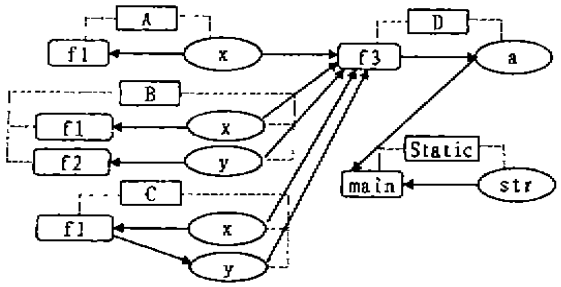

图4 程序1的 MLDG

2.1.3 方法级依赖图(Mthod Level Dependence Graph) Java 的类内部成员函数和成员变量可以通过以下两种情况发生依赖关系：(1)多个成员方法共享某个(某些)成员变量而形成的直接定义或使用关系，简称 MSV(Methods Sharing Variables)关系；(2)通过调用其它方法来间接使用成员变量而形成调用关系，简称 CRV(Call Relation Variables)关系。类依赖图刻画类成员变量和成员函数之间的 MSV 和 CRV 两种关系。而 CRV 可以通过 MSV 来反映。此时我们把方法的参数和返回值看作特殊的成员变量。而方法中定义或使用的成员变量也被看作方法的隐含参数。如果方法 m1调用了方法 m2，则 m1对 m2中的参数变量(包括隐含参数)进行了定义，同时，m1使用了 m2的返回值。以下提到的变量指成员变量或参数变量(包括隐含参数)。

方法级依赖图的生成算法描述如下：

1)如果一个类的方法 M 使用了某个变量 i，则从 M 到 i 有一条使用边。

2)如果一个类的方法 M 定义了一个变量 i，则从 i 到 M 有一条定义边。

3)如果类 C1的方法 M1可能调用类 C2的方法 M2，则 C2的所有子类的 M2方法(可能被覆盖)的参数(包括隐含参数)有一条定义边到 M1。

程序1的 MLDG 如图4所示，其中，类 B 和 C 中的从类 A 继承过来的方法 f1被它们各自的方法所覆盖，图中不再画出。

2.1.4 语句级依赖图 关于这一级依赖图的讨论已经有很多，如 SDG[5]，OSDG[1]，实际我们对 SDG 进行了新的扩展，用来表示语句间的依赖关系，以便得到更为精确的程序切片。

2.2 切片生成层

2.2.1 包级切片 在一个较高的层次上首先去除掉与(v,i)无关的包，以减少后面的切片时的运算量。

包级切片算法如下：1)根据切片准则(v,i)定位切片点 i 所在的包 p，2)从代表 p 的结点出发，根据 PLDG 找出所有由"引入边"直接到达或间接到达 p 的结点，3)这些结点的集合就是程序关于(v,i)的包级切片 S′(P)。

2.2.2 类级切片 在包级切片的基础上，去除掉一些无关的类。类级切片算法：

4)根据包级切片 S′(P)在 CLDG 中将无关的结点去除掉。

5)根据切片准则(v,i)定位切片点 i 所在的类 c，然后根据 CLDG 找出 c 结点沿创建边能够到达的结点集合 I。

6)根据 CLDG 找出从 I 中的结点出发沿继承边可以到达的结点集合 I1，I 和 I1的并集就是要找的关于(v,i)的类级切片 S″(P)。

考虑对14行的变量 a.x 进行类级切片。从其 CLDG 中的 B 结点出发，沿创建边能够到达的结点集合 I 为{B、D 和 Static}。然后从 I 中每个结点出发，沿继承边能到达的集合为{A}，而 C 结点被去除了。

2.2.3 方法级切片 在类级切片的基础上进一步去除掉无关的类和方法。

方法级切片算法：由于基类的方法和成员变量会被子类继承，因此，首先根据切片准则(v,i)定位切片点 i 所在的方法的集合 M0，v 也可能代表多个变量的集合 V0，然后根据 MLDG 找出和 m 结点有 MSV 和 CRV 关系的结点。算法如下：

1)根据类级切片第2)步，去除掉不属于集合 I 中的类；
2)M＝M0；V＝V0；
3)if M＝{}then 算法结束；
For M 中的每一个结点 m1
找出由 m1出发，有使用边到达的未标记变量结点的集合 V1；V＝V or V1；
End for
标记所有属于 M 的结点；M＝{}；

4)If V={}then 算法结束;

For V 中的每一个结点 vl

找出由 vl出发,有定义边到达的未标记变量结点的集合 M1;M=M or M1;

End for

标记所有属于 V 的结点;V={};

5)回到第2)步。

算法结束后,所有被标记的变量和方法的集合就是要找的关于(v,i)的方法级切片 $S^{m}(P)$。例如,考虑对14行的变量 a.x 进行方法级切片,首先根据类级切片的结果集合 I 在 MLDG 中(如图4所示)去除掉类 C 和类 A 的方法和变量结点。M0中只包含 D 的 f3结点,记做 D_f。V0本来包含 A_x、B_x 和 C_x,然而 C 和 A 都已经被去除了。然后从其 CLDG 中的类 D 的 f3结点,记做 D_f3,开始回溯,有定义边到达 f3的变量结点有 D_a,再从 D_a、B_x 回溯,有使用边到达的结点有 B_f1 和 Static_main。于是得到方法级切片{Static_main,D_f3,B_f1,D_a1,B_x}。

2.2.4 *程序切片* 依赖于语句级依赖图的表示方法。实践中,仍然采用两步可达性算法[5]。

2.3 另外一种体系结构

在我们的模型中依赖图全在没有生成切片之前生成。然后进行切片。这对我们对同一组 Java 源程序多个切片时是有利的。如果对很多组程序进行一个切片时,模型采取另一种体系结构效率将更高。即,生成 PLDG 后立即进行包级切片,然后生成 CLDG,此时的 CLDG 中将不包含一些包中的类,其复杂度将减少。同样在得到类级切片后再生成 MLDG,在得到方法级切片后再生成语句级依赖图,这样可以减少 MLDG 和语句级依赖图的复杂度。

3 比较

3.1 切片结果的比较

在以往的工作中,对多态的处理显得较为无力。特别是多态对象参数(如程序1中,22行的参数 a)。D. Liang[4]提出了一种处理方法,但是用这种方法得出来的切片不够精确。如用这种方法对14行的变量 a.x 进行类级切片,将会把类 C 和类 A 中的方法也包括进去。而分层切片模型能够在类切片级和方法级就将一些无关的类和方法排除。只要选用恰当的语句级依赖图和切片算法,不难产生更为精确的切片。

3.2 算法复杂度的比较

我们的模型是同时生成各种依赖图。语句级依赖图可以采用以前的方法生成,其复杂度不变。而包级、类级、方法级依赖图的复杂度都远小于语句级依赖图。所以算法的复杂都只是略为增加。而如果采取2.3节中提到的另一种体系结构,由于语句级依赖图结点个数减少了,整个算法的复杂度可能比传统的方法还要小。

结论 本文介绍了分层切片工具模型 JSTM 的设计思想和实现:通过构造 PLDG、CLDG、MLDG 和语句级依赖图逐步生成包级切片、类级切片、方法级切片和程序切片,并对这种算法与传统的算法进行了一些比较。

参考文献

1 Krishnaswamy A. Program Slicing: An Application of Object-oriented Program Dependency Graphs [Technical Report TR94-108]. Department of Computer Science. Clemson University, 1994

2 Zhao J J, Cheng J D, Ushijima K. Static slicing of concurrent object-orienteed programs. http://www.fit.ac.jp/~zhao/paper.html

3 Larsen L D, Harrold M J. Slicing Object-Oriented Software ICSE96, 1996. 495～505

4 Liang D, Harrod M J. Slicing Objects Using System Dependence Graphs. In: Intl. Conf. on Software Maintenance, Nov 1998

5 Horwits S, Reps T, Binkley D. Interprocedual Slicing Using Dependence Graphs. ACM Transacteons on Programming Languages and System, 1990, 12(1): 26～60

6 Ottenstein K J, Ottenstein L M. The program dependence graph in a software development environment In: Proc. of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Sotrware Development Environments (Pittsburgh, Pa, April 23-1984). ACM SIGPLAN not. 19, 5(May 1984), 177～184

7 Zhao J J. Dynamic slicing of object-ortented program: [Technical Report SE-98-119]. Information Processing Society of Japan, May 1998. 17～23

8 李必信,等. 一种面向对象程序的分层切片方法. 软件学报. 已录用,待发表

9 刘小东,等. OOPSE——一种基于 C++/Java 的程序分析. 计算机科学, 2001, 28(1)

10 张勇翔,等. 程序切片技术及其应用. 计算机科学, 2000, 27(1)