类的数据流分析方法研究

孙跃勇　张　涌　赵文耘　丁　文
（复旦大学计算机科学与工程系　上海　200433）

摘　要　本文把类的操作划分成三个不同的级别，采用增量算法分别对不同级别的操作进行数据流分析，相应得到三种不同的定义－引用对，根据生成的定义－引用对就可以进行基于数据流的测试。如何对有调用关系的操作之间进行快速有效的数据流分析是对类进行数据流分析的一个难点，本文对该问题进行了深入研究，并提出了相应的解决方法。

关键词　数据流分析　定义－引用对　操作级别　操作调用序列　操作调用图　控制流图

RESEARCH ON DATA FLOW ANALYSIS APPROACH FOR CLASS

Sun Yueyong　Zhang Yong　Zhao Wenyun　Ding Wen
(*Department of Computer Science & Engineering, Fudan University, Shanghai 200433*)

Abstract　The basic unit of testing on object－oriented software is class. In order to perform data flow analysis easily on class, in this paper we classify class methods into three levels, then incremental algorithms are adopted to analyze the data flows. According to these incremented data flow analysis algorithms, we get three kinds of def－use pairs, after that, data flow testing can be performed based on those def－use pairs. The focus of this article is on quick and effective data flow analysis between class methods which have invocation associations and produce def－use pairs for public methods.

Keywords　Data flow analysis　Def－use pair　Method level　Method invocation sequence　Method invocation diagram　Control flow graph

1　引　言

面向对象的软件开发方法已经取代了传统的面向过程的方法，成为当今软件开发技术的主流方法。类是面向对象方法的基本单元。类的实例就是对象。类把描述对象特征的属性和对这些属性的操作封装在一起，实现了信息隐藏。面向对象的另一个重要特性是继承，子类可以继承父类的属性和操作，提高了代码重用率。因此类也是面向对象程序测试的基本单元。类的信息隐藏、继承、多态等特性对软件测试提出了新的挑战。

现在对类进行测试主要有两种方法：基于对类的规约说明的方法和数据流分析方法。大多数基于规约说明的方法[5,6]都要求对类有很好的形式化描述，并根据描述设计测试用例，比如 Didier Buchs 等人提出的面向对象的并行描述语言 CO－OPN/2。但很多软件的描述很不完善，甚至没有任何形式化的描述，对这些软件我们就无法应用基于状态描述的方法进行测试。并且基于状态描述的方法是黑盒测试法，不能根据某种代码覆盖策略生成相应的测试用例来实现充分的代码覆盖，这就要求用白盒测试方法来配合进行充分有效的测试。数据流分析就是一种基于代码的白盒测试法。传统的面向过程的数据流分析技术包括单个过程的数据流分析和过程间的数据流分析。前者解决的是对单个过程内部的变量进行数据流分析，但对于全局变量、引用参数、变量别名这样可能穿越过程边界的变量就无法对其进行分析；Harrold 等人提出了过程间数据流分析方法成功地解决了对穿越过程过界的变量进行数据流分析的问题[1]。

Harrold 和 Rothermel 把传统的数据流分析方法应用到类的测试上，他们拓展了 Pande, Landi 和 Ryder 的 PLR 算法[1]，提出了对类进行数据流分析的框架[2]，该框架可以对该类的任意过程调用序列进行数据流分析。可是他们却没有说明如何具体地对类进行数据流分析，没有指出怎样从类的源代码得到属性的定义－引用对。

本文提出了一个具体的对类进行数据流分析的算法，利用该算法可以方便地对类的属性进行数据流分析，且易于计算机实现。该算法把类的操作分成三个

收稿日期：2001－05－24。孙跃勇，硕士生，主研领域：软件复用和软件测试。

由低到高的级别,对不同的级别采用不同的分析方法,对较低级别的操作执行数据流分析的结果为分析较高级别的操作提供了必要信息,因此是一种增量数据流分析算法。

2 数据流分析和测试

2.1 基本概念

数据流分析是通过对变量构造定义 - 引用对来实现的。定义 - 引用对是一个有序对(d,u):设 v 是程序中某一变量,d 是在程序中对变量 v 进行赋值(定义)的语句,u 是程序中引用变量 v 的语句,并且从定义点 d 存在一条执行路径可以到达引用点 u,则称有序对(d,u)是变量 v 的一个定义 - 引用对,下面出现的(d,u)如不加详细说明都指变量 v 的定义 - 引用对。

利用变量 v 的(d,u)可以设计出相应的测试用例,使得程序的执行路径可以通过这个(d,u)对,从而对程序进行数据流测试。根据选定的覆盖标准生成合适的测试用例集就可以实现对程序的数据流覆盖。

在设计测试用例时要对定义 - 引用对进行有效性分析,保留有效的(d,u)对,抛弃无效的(d,u)对。有效的(d,u)对是指从变量 v 的定义点 d 开始存在一条可执行路径可以到达 v 的引用点 u,并且在该路径上不存在对变量 v 的重新赋值,即是一条干净的路径。数据流测试只对有效的(d,u)对生成测试用例。

2.2 传统的数据流分析技术

传统的面向过程的数据流分析分为过程内的数据流分析和过程间的数据流分析。过程内的数据流分析:通过构造过程的控制流图 CFG 来对过程进行基本块的划分;结点代表每个基本块,边代表了控制流,每个 CFG 都对应唯一的一个入口结点和出口结点;根据 CFG 就可以分析出变量的定义 - 引用对。但是这种方法只能处理局部变量,对程序中的全局变量和引用参数就无法进行数据流分析,因此引出了过程间的数据流分析。[1]中提出了一个过程间数据流分析的方法,我们叫它 PLR 算法。PLR 算法构造了一个过程间的控制流图 ICFG,在每个过程调用点把调用语句替换成 call 结点和 return 结点,并通过边把 call 结点和被调用过程的 entry 结点连结起来,同样把被调用过程的 exit 结点和 return 结点相连结。用一个特殊的 main 结点来表示 main 函数的 entry 结点,表示是整个程序的 entry 结点。这样就把整个程序的过程连结到一起,从而可以进行过程间的数据流分析。

3 类和操作的级别

3.1 类

类是属性和操作的封装体,属性代表了对象的本质特征,对象的一组属性值决定了对象当前的状态,调用对象的操作取得什么样的结果也要根据对象当前状态而定。因此对类的属性进行数据流分析是十分重要的。

在 C++ 类中,对类的属性和操作提供了几种不同的访问级别:公有的、私有的和受保护的。公有的属性和操作可以由类的使用者任意访问,对外部是完全公开的;私有的属性和操作则不允许外部访问;受保护的属性和操作可以由该类本身及其子类来访问,其它外部使用者没有访问权限。每个类都有一个或多个构造函数和析构函数,每当为该类生成一个实例的时候就调用构造函数,当销毁这个实例时就自动调用析构函数。

让我们看下面一个 C++ 类的例子,它实现了一个自动售货机的功能:

```
Class CoinBox{
    private:
      unsigned int totalQtrs;   //total quarters collected
      unsigned int curQtrs;     //current quarters collected
      unsigned int allowVend; //1 = vending is allowed
      unsigned int IsAllowVend(){return allowVend;}
      void Reset(){totalQtrs = 0;curQtrs = 0;allowVend = 0;}
    public:
      CoinBox(){Reset();}
      void returnQtrs(){curQtrs = 0;}//return current quarters
      void addQtrs(){
        curQtrs = curQtrs + 1; //add a quater
        if(curQtrs > 1)         //if more then one quarter is
          allowVend = 1;        //collected,then set allowVend
      }
      void vend(){
          if (IsAllowVend()){   //if allowVend
            totalQtrs = totalQtrs + curQtrs;//update totalQtrs
            curQtrs = 0;                    //update curQtrs
            allowVend = 0;                  //update allowVend
          }
      }
}
```

图 1 类 CoinBox

类 CoinBox 中有三个属性:totalQtrs, curQtrs 和 allowVend。属性 totalQtrs 记录了这台自动售货机中的硬币总数;属性 curQtrs 记录了自从上次销售后用户又投入的硬币数量,并规定至少投入两枚硬币才可以出售货物;属性 allowVend 表明用户为购买货物是否投入了足够的硬币。

3.2 操作的级别

为了清楚地表明操作之间的调用关系,我们给出了 CoinBox 类的操作调用图(如图 2 所示)。图中的矩形结点表示类 CoinBox 的操作,操作前面的'+'表示该操作是公有操作、'-'表示是私有操作;矩形结点之

间带实心箭头的实线表示操作之间的调用关系;外面大的矩形表示整个类的边界,从边界出发指向内部矩形结点的带空心箭头的线表示类的外部使用者可以以任意序列调用这些结点代表的操作,显然这些操作都是公有操作。

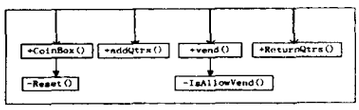

图2 CoinBox类的操作调用图

指向CoinBox()结点的线是虚线,表示该结点代表的操作是类的构造函数。构造函数和析构函数用粗虚线表示他们不同于其他的公有操作,构造函数在创建对象时调用,析构函数在销毁对象时调用。其他的公有操作则可以在对象生存期间由外部使用者随意调用。由类的操作调用图可以把类的操作分为三个级别:

定义1 Server级:处于这个级别的操作不调用类的任何其它操作,它只被其它操作或者类的外部使用者所调用,即只提供调用服务。Reset()和IsAllowVend()属于Server级。

定义2 Client级:该操作调用了类的其它操作,被调用的操作既可以是Server级别的操作、又可以是Client级别的操作,因此是一个递归定义。CoinBox()和vend()属于Client级,因为它们分别调用了Reset()和IsAllowVend()。

定义3 Interface级:类的外部使用者可以调用该操作,即是类的公有操作。addQtrs()、ReturnQtrs()、vend()是公有操作,因此属于Interface级,CoinBox()是构造函数同时也是Interface级操作。

一个Client级或Server级的操作可以是Interface级操作,如:vend()调用了操作IsAllowVend(),因此是Client级,它又是公有操作,所以也是Interface级;addQtrs()没有调用其它操作,所以是Server级操作,同时它是公有操作,因此也就属于Interface级。

把类的操作划分成不同的级别,从而可以对不同级别的操作采用不同的策略进行数据流分析。对Server级别的操作:因为它们不调用任何其它操作,即不存在和其它操作的关联,因此应最先进行数据流分析;对Client级别的操作:因为调用了其它操作,即和被调用的操作有关联,有可能涉及到操作间的数据流动,所以应在被调用的操作之后进行数据流分析;Interface级别的操作可以由类的外部使用者以任意序列调用,也就是说,调用序列是不确定的,因此只能给出Interface级别的操作之间可能的定义-引用对。

4 类的数据流分析

对应前面的三个不同的操作级别,我们给出三种定义-引用对。为方便起见,在下面的叙述中,我们假定C是要测试的类(CUT:Class Under Test);v代表类C的某个成员变量,即属性:d表示定义v的语句;u表示引用v的语句。

定义4 Server操作定义-引用对(d,u):假定M_{ser}为C的Server操作,即它没有调用C中其它操作,如果d和u都在M_{ser}内部,且存在一条执行路径可以从d到达u,则称(d,u)是Server定义-引用对。

定义5 Client操作定义-引用对(d,u):假定M_{cli}是C的Client操作,$\{M_1,M_2,\cdots,M_n\}$是M_{cli}所调用的操作序列(可以是直接调用,也可以是间接调用),其中$M_i(1=<i<=n)$是C的操作。如果存在M_i包含d,M_j包含u($i<>j$),且M_i、M_j属于$M_{cli}\cup\{M_1,M_2,\cdots,M_n\}$,又v在d点的值可以传播到u的话,则称(d,u)是Client定义-引用对。M_i和M_j($i<>j$)可以是同一个操作,因为在M_{cli}中可能多次直接或间接调用C的某个操作。

定义6 Interface定义-引用对(d,u):假定M_{int}是C的公有操作,$\{M_1,M_2,\cdots,M_n\}$是M_{int}所调用的操作序列;假定N_{int}是C的公有操作,$\{N_1,N_2,\cdots,N_n\}$是N_{int}所调用的操作序列;如果$\{M_{int},M_1,M_2,\cdots,M_n\}$中存在d,$\{N_{int},N_1,N_2,\cdots,N_n\}$中存在u,则称(d,u)是Interface定义-引用对。

4.1 对Server级操作进行数据流分析

设M_{ser}为C的操作,v为C的属性,M_{ser}属于Server级,即没有调用C的其它操作。M_{ser}对v进行了赋值和引用。

4.1.1 M_{ser}内部的定义-引用对

通过研究传统的面向过程的数据流分析方法,我们发现其实传统的对单个模块进行数据流分析的方法适合于类的Server级操作的数据流分析,因为Server级的操作不存在和其它操作的关联,故可以看作独立的单元。只要把属性v看作是全局变量,对M_{ser}的数据流分析就可以利用传统的对一个独立的单元进行数据流分析的方法来实现。具体实现可以利用传统的控制流图对M_{ser}划分基本块(可以参看[4]),从而得到M_{ser}中的定义-引用对的集合<v,d,u>:v是C的属性,d是M_{ser}中对v的定义点,u是M_{ser}对v的引用点,且从d存在一条可执行的路径到达u,在该路径上不存在对v的其它定义点。同定义4可知,<v,d,u>是Server级别的定义-引用对。构造出定义-引用对以后,可以相应地生成测试用例对M_{ser}进行数据流测试,这类似于传统的单元测试技术。

4.1.2 构造集合$M_{ser}_In(v)$,$M_{ser}\_Out(v)$

因为M_{ser}属于Server级操作,它或者可以被类的外

部使用者调用，或者被 C 的其它操作调用，为了进行集成测试，需要构造两个集合 $M_{ser}\_In(v)$，$M_{ser}\_Out(v)$：

定义 7 $M_{ser}\_In(v)$：表示 M_{ser} 内部对 v 的引用点的集合，且在这些引用点之前不存在对 v 的重新赋值，也就是说属性 v 在 M_{ser} 外部的定义值可以到达的 M 的内部引用点。

定义 8 $M_{ser}\_Out(v)$：表示 M_{ser} 内部对 v 的定义点的集合，且在这些定义点之后不存在对 v 的重新赋值，即对 v 在 M_{ser} 内部定义，并能够传播到 M_{ser} 外部的那些定义点。

$M_{ser}_In(v)$的构造算法：采用深度优先或广度优先的算法去遍历操作 M_{ser} 的各个分支。若遇到分支则把分支条件记录下来，并和前面的分支条件进行与操作；如果在某条路径上发现对 v 的定义则立即中止在该路径上的搜索，根据选定的算法回溯到前一个结点继续搜索，直到所有的路径都遍历完毕，即可构造出 $M_{ser}\_In(v)$。

$M_{ser}_Out(v)$的构造算法：操作 M_{ser} 对 v 进行了重新定义，v 的值会传播到 M_{ser} 外部，因此要记录下 M_{ser} 内部对 v 的有效定义点，即其值可以穿过出口节点到达 M_{ser} 外部的 v 的定义点。

分析 M_{ser} 的控制流图，从入口结点开始进行搜索，在每个执行路径上找到离出口结点最近的定义点，即有效定义点。

$M_{ser}_In(v)$中的元素用如下形式来表示：

< v, precondition, position >

其中 position 是引用属性 v 的语句标号；precondition 是从 M_{ser} 的入口结点到达 position 的分支条件，如果从入口到 position 只有一条执行路径、不存在其他分支，则 precondition 为 TRUE。

$M_{ser}_Out(v)$中的元素表示形式如下：

< v, precondition, new value, position >

其中 position 是属性 v 定义点的语句标号，precondition 是从 M_{ser} 的入口结点到 position 的分支条件，new value 是在 position 定义点对 v 的定义值。

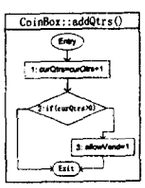

图 3 控制流图

我们采用图 1 中的 addQtrs()操作为例，由图 2 可知它是 Server 级的操作，因为它没有调用其它的操作，我们首先画出它的控制流图(见图 3)。

由图 3 的控制流图可以得到 addQtrs()的 addQtrs_In(v)。

v	Precondition	Position
curQtrs	TRUE	1

同样可以得到 addQtrs()的 addQtrs_Out(v)：

v	Precondition	New value	Position
curQtrs	TRUE	curQtrs′ + 1	1
allowVend	curQtrs′ + 1 > 0	1	3

curQtrs′是指在 addQtrs()入口时的值，因为 curQtrs 在 addQtrs()中被重新定义，其值已经发生改变，所以用 curQtrs′来表示其原来的值。

4.1.3 $M_{ser}_In(v)$的优化

构造 $M_{ser}_In(v)$的目的是为了方便地进行操作间的集成数据流分析，最终是为了生成测试用例进行数据流测试。

在基于数据流的测试中，测试覆盖的标准是覆盖每个定义－引用对，在遵循这个标准的前提下应力求构造最少数量的测试用例。

传统的数据流分析方法是先生成所有可能的定义－引用对，然后找出最少的测试用例集。我们在构造定义－引用对的时候，就考虑到了如何简化测试用例，在遵循覆盖标准的情况下构造出最少的定义－引用对，这样大大减少了测试用例的数目，而且易于计算机实现。

为此我们引入了同心引用点的概念。

定义 9 同心引用点：通过上述的 $M_{ser}_In(v)$的构造算法得到的引用点集合中，如若存在某属性 v 的 n ($n>1$)个引用点$\{S_1, S_2, \cdots, S_n\}$，它们的 Precondition 相同，则称这 n 个引用点为 v 的同心引用点。

在以前的数据流测试技术中，为了上述的 n 个同心引用点要生成 n 个测试用例$\{T_1, T_2, \cdots, T_n\}$，使得 T_i 可以覆盖 S_i ($1 = <i< = n$)。

由同心引用点的定义可知，各个引用点 S_i 的分支条件相同，即$\{S_1, S_2, \cdots, S_n\}$在同一条执行路径上，换言之，如果执行某个测试用例 T_i 可以覆盖 S_i，则 T_i 同样可以覆盖$\{S_1, S_2, \cdots, S_n\}$中任意一点，因此只需保留一个测试用例即可。

故而在利用搜索算法构造出原始的 M_In(v)后，要对其进行优化。具体方法是针对各个属性找出同心引用点$\{S_1, S_2, \cdots, S_n\}$ ($n>1$)，保留 S_1，抛弃$\{S_2, \cdots, S_n\}$。通过优化既可以减少测试用例的数目，同时又不降低覆盖率。

4.2 对 Client 级操作进行数据流分析

假定 M_{cli} 是 C 的 Client 级操作，$\{M_1, M_2, \cdots, M_n\}$是 M_{cli} 所直接或间接调用的操作序列。M_{cli} 调用了 M_i ($1 = <i< = n$)，因此在对 M_{cli} 进行数据流分析时要把 M_{cli} 和$\{M_1, M_2, \cdots, M_n\}$集成到一起进行分析，这就涉及到了操作间的数据流分析技术。

不失一般性，我们假定 M_{cli} 直接调用了 Server 级操作 M_1，设 S 为调用点，并且已经对 M_1 进行了数据流分

析。利用前面对 Server 级操作相同的算法来对从 M_{cli} 的入口到 S 的一段代码进行数据流分析，并构造属性 v 的 $M_{cli}\_In(v,S)$、$M_{cli}_Out(v,S)$集合和 Active(v,S)，其定义如下：

定义 10 $M_{cli}\_In(v,S)$：在操作 M_{cli} 中可以到达 S 的路径上对属性 v 的引用点集合，并且该路径上不存在对 v 的定义点。

定义 11 $M_{cli}\_Out(v,S)$：在操作 M_{cli} 中可以到达 S 的路径上对属性 v 的定义点集合，并且其定义的 v 的值在 S 点是活跃的。

所谓变量 v 在某点 S 是活跃的是指变量 v 的值可以沿着某条路径传播到 S 点。

定义 12 Active(v,S)：返回一个布尔值，若为真表示沿着 M_{cli} 的入口结点到对 M_1 的调用点 S 的路径中有一条干净路径，即在这条路径上不存在对属性 v 的定义点；若为假则表明不存在这样的干净路径。

如果在 M_{cli} 中某点 d 对 v 进行了定义，且 d 在对 M_1 的调用点 S 之前，即 $M_{cli}_Out(v,S)$不为空，又 $M_1_In(v)$不为空，即 d 对 v 的定义可以传播到 M_1 中对 v 的引用点，则根据 $M_{cli}_Out(v,S)$和 $M_1_In(v)$就可以构造出定义点在 M_{cli} 中、引用点在 M_1 中的定义－引用对(d,u)，其中 $d\in M_{cli}$，$u\in M_1$，我们记为 $M_{cli}\_M_1(v,d,u)$，以后对 M_{cli} 进行数据流测试用例时就可以根据 $M_{cli}_M_1(v,d,u)$中的定义－引用对生成相应的测试用例。

通过构造的上述三个函数，我们可以对 M_{cli} 和 M_1 进行集成的数据流分析，目的是为了得到 $M_{cli}\_In(v,S')$、$M_{cli}\_Out(v,S')$，其中 S′是 S 的下一条语句。构造 $M_{cli}_In(v,S')$是为了把 M_1 中对 v 的引用点加入到 M_{cli} 中，以便集成分析。同理构造 $M_{cli}_Out(v,S')$也是要把 M_1 中对 v 的定义点加入到 M_{cli} 中。

构造 $M_{cli}\_In(v,S')$

首先分析 Active(v,S)：

1）如果为真：表明从 M_{cli} 的入口结点到 S 存在一条干净路径，不妨设从入口结点沿这条路径到达 S 的分支条件为 P_c。

```
IF M1 _ In(v)为空 THEN  {M1 中没有对属性 v 的引用点}
   IF M1 _ Out(v)为空 THEN  {M1 中没有对属性 v 的定义点}
      Mcli_ In(v,S′) = Mcli_ In(v,S);
   ELSE BEGIN {M1 _ Out (v)不为空，不妨设为 D1,D2,…,Dn}
      {设 P1,P2,…,Pn 分别为 D1,D2,…,Dn 的前置条件(Precondition)}
      IF P1∪P2∪…∪Pn = TRUE THEN
         Mcli_ In(v,S′) = Mcli_ In(v,S);
   END
```



```
ELSE BEGIN {M1 _ In(v)不为空，不妨设为 U1,U2,…,Un}
   {设 C1,C2,…,Cn 分别为 U1,U2,…,Un 的前置条件(Precondition)}
   Mcli_ In(v,S′) = Mcli_ In(v,S);
   WHILE Ci ≠ TRUE DO    ①
      Mcli_ In(v,S′) = Mcli_ In(v,S′) + {Pc∩Ci | Ui}
   END
   IF P1∩P2∩…∩Pn = TRUE THEN STOP Mcli_ In (v,S′)    ②
   ELSE PC: = Pc∩(NOT P1∪P2∪…∪Pn)    ③
```

① 能到达 S 的测试用例一定可以到达 $M_1_In(v)$中前置条件为 TURE 的引用点。

② 如果 $M_1_Out(v)$的所有出口都对属性 v 进行了定义，则结束对 $M_{cli}_In(v,S')$在 M_c 中的构造。

③ 在 M_1 中存在一条 v 的干净路径，则构造穿越 M_1 到达 S′的分支条件 PC，以备下面继续分析使用。

2）如果不为真：沿着 M_{cli} 的入口结点到对 M_1 的调用点 S 的路径中不存在一条干净路径，则 $M_{cli}\_In(v,S')=M_{cli}\_In(v,S)$，构造过程结束。

构造 $M_{cli}-Out(v,S')$

设从 M_{cli} 的入口结点到 S 的路径的分支条件是 P_c，$M_1\_Out(v)=\{D_1,D_2,\cdots,D_n\}$，不妨设 $P_1,P_2,\cdots,P_n$ 分别为 $D_1,D_2,\cdots,D_n$ 的前置条件(Precondition)。

```
IF  Mcli_ Out(v,S)为空 THEN
   Mcli_ Out(v,S′) = {Pc∩Pi | Di};
ELSE {Mcli_ Out(v,S)不为空}
Mcli_ Out(v,S′) = {Pc∩Pi | Di} + {P′c∩(NOT P1∪P2∪…∪Pn) | D′i};
```

其中 $D'_i\in M_{cli}\_Out(v,S)$，P'_c 是 D'_i 的前置条件。

4.3 任意操作调用序列的数据流分析

假定类 C 有 n 个公有操作$\{M_1,M_2,\cdots,M_n\}$，通过前面的步骤，对任意的 C 的属性 v 我们已经构造出 $M_i_In(v)$和 $M_i_Out(v)$以及 M_i 内部的定义－引用对(其中 $M_i\in\{M_1,M_2,\cdots,M_n\}$，因此我们可以通过它们构造出整个类 C 的 DU 对集。

表示形式如下：

<variable, M_d _ Pre, M_d, M_u _ Pre, M_u>

variable 是要构造 DU 对的属性，M_d 是包含 variable 的定义点的操作，M_d_Pre 是由 M_d_Out(variable)中得到的该定义点的 Precondition；同样 M_u 是包含 variable 的引用点的操作，M_u_Pre 是由 M_u_In(variable)中得到的该引用点的 Precondition。

生成类 C 的 DU 对的集合的算法如下：

```
FOR 每个 v∈V(C) DO{其中 V(C)是类的所有属性的集合}
BEGIN
   FOR i = 1 TO n DO
```

（下转第 43 页）

交换算法，并在系统中实现。随着 DSPS 系统的日益复杂，密钥管理还将会面对更大的挑战。

参 考 文 献

[1] CAI Liang, YANG Xiao - Hu, DONG Jin - Xiang, Thread Analysis and Security Protection for Malicious DMBS, ICYCS 2001.

[2] S. Bakhtiari, R. Safavi - Naini and J. Pieprzyk, "Keyed Hash Functions", Cryptography: Policy and Algorithms, Sprinter - Verlag, 1996, pp. 201 ~ 214.

[3] Colin Boyd, "A Class of Flexible and Efficient Key Management Protocols" IEEE, 1996.

[4] S. M. Bellovin and M. Merrit, "Limitations of the Kerberos Protocol" Winter 1991 USENIX Conference Proceedings, USENIX Association, 1999, pp. 253 ~ 267.

[5] Douglas Maughan, Mark Schertler, Mark Schneider, and Jeff Turner, Internet Security Association and Key Management Protocol (ISAKMP). Internet - draft, IPSEC Working Group, June 1996.

[6] C. Meadows, Applying formal methods to the analysis of a key management protocol, Journal of Computer Security, 1(1)5 - 36, 1992.

[7] A. Aziz, M. Patterson, and G. Baehr, Simple Key - Management for Internet Protocols (SKIP), In Proceedings of the Internet Society International Networking Conference, June 1995.

[8] R. Rivest, The MD5 Message Digest Algorithm RFC 1321 April 1992, Networking Group. MTT Laboratory for Computer Science and RSA Data Security Inc.

(上接第 10 页)

```
BEGIN{找到所有的定义点}
  IF Mi _ Out(v) < > NULL THEN
    把Mi _ Out(v)加入到 OUT(V)   //数组,存放 Mi _ Out(v)
END
FOR i = 1 TO n DO
BEGIN{找到所有的引用点}
  IF Mi _ In(v) < > NULL THEN
    把Mi _ In(v)加入到 IN(V)     //数组,存放 Mi _ In(v)
END
FOR 对每个 def∈OUT(V) DO
  FOR 对每个 use ∈IN(V) DO
    构造(def,use)
END
```

在进行数据流分析并构造 DU 对时，最重要的一点是找出有效的 DU 对、抛弃无效的 DU 对。所谓有效是指在变量 v 的定义点 d 和引用点 u 之间不存在其它 v 的定义点，即从 d 到 u 存在一条干净的路径。

文献[3]也提到了如何通过数据流分析来构造 CUT 的 DU 对，但他的方法是通过画出类的控制流图 CCFG 来手工构造的，而且在构造的 DU 对中存在无效的 DU 对。如何剔除这些无效的 DU 对就成了很重要的问题，因为我们无法为一个不可能执行的路径设计一个测试用例。

[3]中只是通过人工判断来确定哪些是有效的 DU 对，哪些是无效的，不能自动实现。我们的方法是首先构造各个操作的 M_i _ In(v)和 M_i _ Out(v)，在构造的过程中就避免了出现无效 DU 对的可能，因此构造出来的 DU 对集都是有效的。

5 总结和将来的工作

根据类的操作之间有无调用关系我们把类的操作划分为不同的数据流分析级别，对于 Server 级别的操作 M_{ser}，我们除了采用传统的面向过程的数据流分析方法得到其内部的定义 - 引用对之外，还要构造出其集合 M_{ser}_ In(v)和 M_{ser}_ Out(v)，以便对调用它的其它操作进行数据流分析。

另外我们给出了如何对 Client 级别和 Interface 级别的操作进行数据流分析的增量算法，其中每一步都利用了前一步的分析结果，因此大大简化了数据流分析步骤，减少了计算复杂度，分析结果以表格的形式进行存储，可以方便地由计算机来实现。

继承是面向对象中的重要性质，我们的方法只是涉及到一个不考虑继承的简单类，如何对一个有继承关系的子类进行有效的数据流分析是我们下一步的研究目标。

参 考 文 献

[1] M. J. Harrold and M. L. Soffa, Interprocedural data flow testing. In Proceedings of the Third Testing, Analysis and Verification Symposium, pp. 158 ~ 167, December 1989.

[2] M. J. Harrold and G. Rothermel, Performing data flow testing on classes. In 2nd ACMSIGSOFT Symposium on the foundations of software enginering, pp. 154 ~ 163, New Orleans, LA(USA), December 1994.

[3] Ugo Buy, Alessandro Orso, Mauro Pezze, Automated Testing of Classes. In: Proc. of the International Symposium on Software Testing and Analysis, 2000, Portland, pp. 39 ~ 48.

[4] 陈火旺、钱家骅、孙永强，编译原理，国防工业出版社.

[5] Kai H. Chang, Shih - Sung Liao, Stephen B. Seidman, Testing object - oriented programs: from formal specification to test scenario generation.

[6] S. Barbey, D. Buchs, and C. Peraire, A theory of specification - based testing for objectoriented software. In Proceedings of EDCC2 (European Dependable Computing Conference), Taormina (Italy), October 1996, Lecture Notes in Computer Science 1150, pp. 303 ~ 320. Springer - Verlag, 1996.

[7] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm, A Logic - Based Approach to Data Flow Analysis Problems. Acta Inf., 35 (6): 457 ~ 504, June 1998.

[8] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa, Refining data flow information using infeasible paths. In Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE'97), pp. 361 ~ 377. LNCS Nr. 1301, Springer - Verlag, September 1997.