基于 UIO 序列的类重要性度量

姜淑娟[1],鞠小林[1,2],王兴亚[1],李海洋[1],张艳梅[1],刘颖祺[1]
(1.中国矿业大学计算机科学与技术学院,江苏徐州 221116; 2.南通大学计算机科学与技术学院,江苏南通 226019)

摘 要: 程序理解是测试和维护大规模面向对象程序的关键,选择程序的关键类优先开展分析是理解程序结构的一个有效的方法.为支持自动识别软件系统中的关键类,本文提出了一种基于 UIO 序列的类重要性度量方法.首先将软件系统抽象为一个以类为转换的有限自动机模型,随后求解该自动机的 UIO 序列,将该序列集合转化为状态转换树.通过递归计算状态转换树的节点复杂度求得类重要度.并在考虑异常传播的基础上改进了算法.最后通过实验验证了算法的有效性.

关键词: 关键类; 有限状态机; UIO 序列; 程序理解
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2015)10-2062-07
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2015.10.027

Measuring the Importance of Classes Using UIO Sequence

JIANG Shu-juan[1], JU Xiao-lin[1,2], WANG Xing-ya[1], LI Hai-yang[1], ZHANG Yan-mei[1], LIU Ying-qi[1]
(1. *School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, Jiangsu* 221116, *China*;
2. *School of Computer Science and Technology, Nantong University, Nantong, Jiangsu* 226019, *China*)

Abstract: Program comprehension is the key to provide insight into large scale object-oriented programs in the testing and maintenance activities. It's a reasonable way to select and start with the key classes. To identify the key classes in a system automatically, we propose a technique to measure the importance of each class based on Unique Input/Output sequence. Firstly, we abstract the software system as a finite state machine model, and then we compute the Unique Input/Output sequence of the finite state machine and converse the Unique Input/Output sequence to a state transform tree by a proposed algorithm. Finally, we traverse the state transform tree to calculate the importance of the classes. The case studies show the effectiveness of our technique.

Key words: key classes; finite state machine; UIO sequence; program comprehension

1 引言

在面向对象的现代软件工程中,类作为软件系统的基本组成单位,是理解和维护软件的主要对象.当前软件规模日趋庞大,传统的软件维护方法面临诸多挑战.如何更有效地分配有限的资源用于系统核心部件维护成为亟需解决的问题.软件中关键类是指那些实现了系统核心功能的类.这些类对程序理解及系统维护起着重要作用.因此,对类的重要性进行恰当的度量,找出软件系统中的关键类,确定程序中的类在软件理解与维护中的优先顺序,可以高效分配软件维护资源.然而,当前对类进行重要性度量的相关研究比较少.截止到目前,周毓明和王木生等人[1]提出了基于类依赖结构模型及 h 指数的类重要性度量方法,从类间依赖关系来计算类之间依赖度,并通过 h 指数来度量类的重要性,Zaidman 等人[2]提出一种基于 Web 挖掘技术的类重要性度量方法,通过将软件系统抽象为有向图,利用搜索引擎中的 HITS[3]算法度量类的重要性.杨芙清等人[4]提出基于类之间关系分析的构件标志算法,用于识别出高质量可复用构件.金茂忠和刘超等人[5]基于类的特性(如继承量、通信量和复杂度)对程序中的类进行度量,给出相应的度量算法并设计实现了一个 Java 程序度量工具.最近,Marios 和 Al Dallal 等人[6,7]提出基于方法层次的类耦合度量方法,用于辅助面向对象程序中类重构及错误定位活动.然而上述方法在实践中受到诸多限制,如需运行并跟踪一组测试用例,从而导致较大的时空开销.由此,我们需要从不同角度探索类重要性度量方法,进一步提高度量的效率和准确率,以此指导软件维护中的程序理

收稿日期:2014-03-03;修回日期:2014-04-14;责任编辑:孙瑶
基金项目:国家自然科学基金(No.61202006,No.60970032);江苏省 333 工程;江苏省高校自然科学研究(No.12KJB520014);江苏省研究生创新工程(No.CXZZ12_0935);南通市应用研究计划(No.BK2012023)

解工作.

另一方面,软件系统可以看作是对数据(如程序输入、内部变量)进行处理的系统.有限状态自动机(Finite State Machine,FSM)是为研究有限内存的计算过程和某些语言而抽象出的一种计算模型[8].它可以表示为一个有向图.有限状态自动机的状态数量有限,且每个状态可以迁移到零个或多个状态,由输入字符串决定状态的迁移.考虑到系统中关键类通常与其他系统类存在着较多的交互,若能够通过度量类的状态变迁图中类之间的状态变迁,进而识别出系统中关键类,将有助于将程序理解工作聚焦于系统中为数不多的关键类.

基于上述分析,本文提出了一种基于有限状态自动机模型的类重要性度量方法.具体而言,在进行类重要性度量时,首先,将系统中的类看作转换,参数和返回值的集合视为状态,从而把软件系统抽象为有限状态机;然后,依据抽象得到的有限状态机求解唯一输入输出(Unique Input/Output,UIO)序列,并将得到的序列集合合并生成系统的状态转换树,通过计算状态转换树的节点复杂度作为对应类的重要度的度量;最后,进一步考虑程序中的异常传播结构,改进节点复杂度计算方法以提高度量结果的精度.对两个大型开源程序的实验表明本文方法可以有效识别程序中的关键类.

2　基本定义

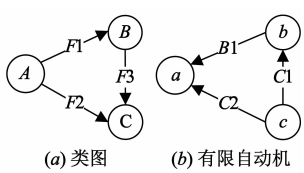
本节介绍与类重要性度量模型相关的定义,随后给出状态树节点复杂度的计算方法.

定义 1　有限状态机是一个五元组:FSM = (I, O, S, δ, λ),其中 I 是输入符号集,O 是输出符号集,S 是状态集,I, O, S 均为非空集合;s_0是初始状态且 $s_0 \in S$;$\delta: S \times I \to S$ 是状态转移函数;$\lambda: S \times I \to O$ 是输出函数.

定义 2　UIO 序列[9,10]表示在自动机模型下,某一状态的唯一的输入输出序列.

例如,状态 S_i的 UIO 序列,记作 UIO_i,是一个特定的从 S_i开始的输入输出序列$\{i_{k,l}/o_{k,l}, \cdots, i_{k,r}/o_{k,r}\}$.其中,$i_{k,l}/o_{k,l}$表示一个输入/输出对.易知,不存在 $S_j \neq S_i$,使得 UIO_i可以从 S_j出发获得.因此,一个 UIO_i序列可以唯一地标识一个状态,但一个状态可以对应多个 UIO_i序列.

定义 3　类转换模型 M 是一个三元组:$M = (N, T, \delta)$,其中 N 是状态集,T 是转换集.$\delta: N \times T \to N$ 是状态转移函数.N, T 均为非空集合.



(a) 类图　　(b) 有限自动机

图1　类转换模型示例

如图 1 所示,某系统 S 中有 3 个类 A、B、C,3 个调用关系 $F1$、$F2$、$F3$,其中 $F1$ 表示 A 调用 B,记为 $A \to B$,$F2$ 表示 A 调用 C,记为 $A \to C$、$F3$ 表示 B 调用 C,记为 $B \to C$.调用的参数记为 $P(F)$,返回值记为 $R(F)$,可得到 S 的有限自动机模型.a、b、c 表示参数与返回值的集合,在模型中为状态.即 $a \subset N$, $b \subset N$, $c \subset N$.其中 $a = R(F1) \cup R(F2)$, $b = P(F1) \cup R(F3)$, $c = P(F2) \cup P(F3)$.$B1(b \to a)$、$C1(c \to b)$、$C2(c \to b)$表示相应的类,即 $B1 \subset T$, $C1 \subset T$, $C2 \subset T$. C 类被调用了两次,相当于 C 对状态做了两次相应的转换,在状态图中分开表示为 $C1$ 和 $C2$.因此,我们可以将一个面向对象的软件系统表示为一个以类集合为转换,参数和返回值为状态的自动机模型.

定义 4　状态转换树(State Transition Tree,STT)是一个二元组:STT = (T, E),其中 T 是节点集合,E 是边的集合,其中边 E_i可以用一个节点对表示:$E_i = < T_{i1}, T_{i2} >$.

通过遍历合并 UIO 序列集合可以得到 STT.例如,$\mathrm{UIO}_i = \{t_1, t_3, t_5\}$,$\mathrm{UIO}_j = \{t_1, t_3, t_7\}$,二者合并得到 $\mathrm{STT}_i = (T_i, E_i)$.其中,下标 i, j 用于区分不同的 UIO 序列,$T_i = \{t_1, t_3, t_5, t_7\}$, $E_i = \{s_1, s_2, s_3\}$, $s_1 = < t_1, t_3 >$, $s_2 = < t_3, t_5 >$, $s_3 = < t_3, t_7 >$.

节点复杂度 $V(T)$由节点 T 的子树复杂度递归定义.其中,T 为状态转换树的节点.如果节点 T 为叶子节点,那么 T 的复杂度 $V(T) = 1$.

$V(T)$的值由式(1)和式(2)进行计算.其中 T_i表示 T 的子树.

$$V(T) = \sum K_i \times V(T_i) \tag{1}$$

$$K_i = V(T_i) \Big/ \sum V(T_i) \tag{2}$$

在状态转换树中,节点复杂度越高表示依赖该节点的转换越多,即该节点出错时引发的错误将会更多.通过计算得到状态转换树的节点复杂度,从而得到相对应的类的重要度.由此解决我们提出的问题.

3　类重要性度量算法

本节给出基于 UIO 序列的类重要性度量方法框架,我们的方法主要分为以下几步:

(1)静态分析 Java 源程序,获取程序结构信息、方法调用信息和方法本身的异常处理结构;

(2)根据程序静态信息构建程序调用图;

(3)分析程序调用图和静态异常信息计算异常传播路径;

(4)根据程序调用图生成类转换模型(状态转换图);

(5)分析状态转换图生成 UIO 序列集合,同时合并为状态转换树;

(6)根据异常信息和状态转换树计算所有节点的复杂度,得到类重要度.

其中前 3 步是预处理阶段(见图 2 中虚线框),该阶段生成的程序调用图和异常传播路径为后面的类转换模型生成和节点复杂度计算提供帮助.后面 3 步为本文主要研究内容,下面将详细介绍这些步骤.

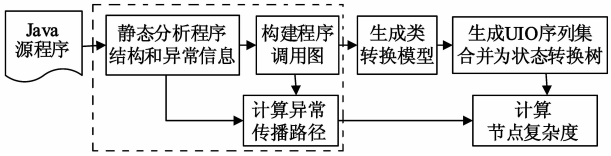


图2 类重要性度量方法整体框架

3.1 生成类转换模型

基于定义 3,我们将软件系统视为类对数据加工转换,把类抽象视为类转换模型中的转换,类之间调用传递的参数及返回值抽象视为数据状态.从而将待分析的软件抽象为类转换模型.具体的,需要将类间调用图转换为状态转换图,详细步骤如下:

(1)获取每个调用信息,包括调用的类,被调用的类以及参数与返回值.例如,图 1 中的 A、B、C 以及 $F1$、$F2$、$F3$ 即为调用信息.

(2)根据类间交互创建相应的状态,状态作为参数与返回值的集合.例如,在图 1 的例子中创建 a、b、c 这 3 个状态,分别是 $F1$、$F2$、$F3$ 这 3 个调用的参数和返回值的并集.

(3)将调用反向记录为被调用类,作为转换名.例如,将 $F1$ 记录为 $B1$,$F2$ 和 $F3$ 分别记录为 $C1$ 和 $C2$.

(4)重复步骤(3)直到遍历完成所有的类.算法最后得到定义 1 所描述的状态转换图.其中变量的集合,包括参数和返回值,是状态集 N,类的集合是转换 T.状态迁移函数在实际中根据系统的具体调用关系分析得到.

3.2 生成 UIO 序列

状态转换图是一个有向图,从一个节点开始通过深度遍历可以得到图中的一条通路.但是通路的集合并不一定能够合并为树,因此有向图不一定能够转换为等价的树.将状态转换图使用树的形式来表示,并使用唯一输入输出序列(即 UIO 序列)来描述.由定义 2 可知,UIO 序列可确定一个唯一的有限状态机.因此,对状态转换图求解类似 UIO 序列的状态转换序列,并使用该序列集描述状态转换图.

得到状态转换图后,通过计算该图的 UIO 序列得到相应的类序列.UIO 序列的计算没有多项式时间的解法[10],已有研究侧重于利用不同的算法求解最短的 UIO 序列.本文通过 UIO 来形式化的描述软件系统的可能转换状态,因此所有可能的 UIO 序列对系统的描述均是有效的,不需要求出最短的 UIO 序列.

算法 1 描述了从状态转化图抽取 UIO 序列的过程.其中,在初始化时将 UIO 序列长度统一置为一个小整数 N(如 $N=10$),然后通过遍历状态转化图的节点求解 UIO 序列.如果无解则将 N 值随机增加一个整数值,并再次求解.实验表明在 N 值较大时,一次求解成功率较高.算法描述如算法 1 所示:首先根据 N 值查找一条以某个节点为起始的路径,然后记录至路径集合中.第二步选择另外一个节点生成一条路径,如果生成的新路径与集合中的任意路径均不相同,那么将之记录至路径集合.第三步判断生成的新路径是否存在于路径集合中.若是,则重新生成.第四步遍历状态转换图所有节点.

算法 1 UIO 序列生成算法 GenerateUIOs

```
输入:状态转换图(有向图) STG
输出:UIO 序列 uio[][]
      初始化序列长度 numLine
      初始化节点总数 sumVector
      初始化 UIO 序列 uio[][]
begin
1     V = STG.getCurrentNode()
2     int n = 0
3     set V as the init node
4     while(n ≤ sumVector)
5          int k = 0
6          while(k ≤ numLine)
7               if(k < numLine)
8                    deep traversal to the adjacent vertex of V
9                    save the sequence to uio[n][k]
10                   k++
11              else
12                   if(uio[v]! = all of uio[])
13                   k++
14              else
15                   back to V
16                   k--
17    traverse to the next node of V
18    n++
19    return uio[][]
end
```

3.3 度量节点复杂度

对抽象得到的类转换模型求解 UIO 序列,通过遍历合并生成状态转换树.如图 1 中示例,将软件系统抽象为类转换模型时,一个类有可能会对应多个转换(C 类对应 $C1$ 和 $C2$ 两个转换).因此在状态转换树中,多个节点可能对应同一个类.对于此类节点需要最后合并其复杂度.

根据定义 4,使用迭代计算的方法求解每个节点的复杂度.由于每个节点的复杂度依赖于其子节点的复

杂度,所以对于状态树的遍历采用后序遍历的方法.求解某节点复杂度的算法描述如算法 2 所示.

在最坏情况下,算法 2 单个节点的算法复杂度为 $O(n^2)$.由于 $K[T]<1$ 恒成立,因此 $V(T)$的值介于其子节点复杂度的最大值与最小值之间.当且仅当其所有子节点不为叶子结点并且其复杂度都相等时,该节点的复杂度与其子节点的复杂度相同.从而避免了状态转换树的结点复杂度始终大于其子节点情况.图 3 描述了一个包含 12 个节点的状态转换树.

算法 2　节点复杂度度量算法 STTraverse

```
输入:状态转换树的节点 STT[i]
输出:单个节点的复杂度 V
      初始化节点复杂度 V[]
      初始化节点系数 K[]
begin
1     node = STT[i]
2     index = node.getIndex()
3     if(node is leaf node)
4          V[index] = 1
5     else
6          while(j≤node.getChildNum())
7               if(childNode[j] is leaf node)
8                    V[index] + = 1
9               else
10                   V_childNode[j] = STTraverse(STT[j])
11                   for(temp from 0 to node.getChildNum())
12                        K[temp] = V_childNode[j]/V_sum
13                        V[index] + = K[temp] × V_childNode[temp]
14    return V[index]
end
```

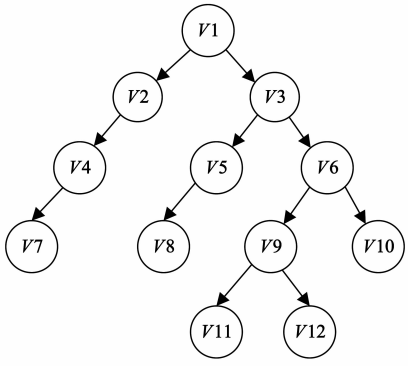


图3　状态转换树

下面以图 3 为例,来说明节点复杂度的计算方法.整个计算过程是对该状态转换树由下而上进行迭代计算各层节点的复杂度:

(1) $V10=1, V11=1, V7=1, V8=1, V12=1$

(2) $V9=V11+V12=2$

(3) $V6=V9/(V9+V10)\times V9+V10/(V9+V10)\times V10=5/3=1.667$

$V4=V7=1, V5=V8=1$

(4) $V3=V5/(V5+V6)\times V5+V6/(V5+V6)\times V6=34/21=1.619$

$V2=V4=1$

(5) $V1=V2/(V2+V3)\times V2+V3/(V2+V3)\times V3=1.382$

3.4　考虑异常的类重要性度量

面向对象程序中的异常处理结构可以有效提高软件的可靠性和健壮性.异常处理结构中包含的代码是经过开发人员精心设计并且认为是可能会出错的代码,因此包含有异常处理结构的类或方法通常是较别的类或方法更为重要.然而异常传播路径中涉及到的方法在异常发生时会将异常传递下去,为了能够刻画异常对状态转换的影响,我们对算法 2 进行改进.具体而言,在分析得到异常传播路径之后,进一步判断当前节点是否在异常传播路径之上.如果状态转换树的节点包含在异常传播路径上,那么在计算其父节点的复杂度时将提升其复杂度,提升的值为常数 C.基于上述分析,对算法 2 描述的算法进行改进,改进后的算法如算法 3 所示.

算法 3　改进的节点复杂度度量算法 STTraverse

```
输入:状态转换树的节点 STT[i],异常路径 ExceptionPath[]
输出:单个节点的复杂度 V
      初始化节点复杂度 V[]
      初始化节点系数 K[]
      初始化常数 C
begin
1     node = STT[i]
2     index = node.getIndex()
3     if(node is leaf node)
4          if(node in ExceptionPath[])
5               V[index] = 1 + C
6          else V[index] = 1
7     else
8          while(j≤node.getChildNum())
9               if(childNode[j] is leaf node)
10                   if(childNode[j] in ExceptionPath[])
11                        V[index] + = 1 + C
12                   else V[index] + = 1
13              else
14                   V_childNode[j] = STTraverse(STT[j])
15                   for(temp from 0 to node.getChildNum())
16                        if(childNode[j] in ExceptionPath[])
17                             childNode[j] = childNode[j] + C
18                        K[temp] = V_childNode[j]/V_sum
19                             V[index] + = K[temp] × V_childNode[temp]
20    return V[index]
end
```

假设图 3 描述的状态转换树中节点 6、9 和 11 包含

在某条异常传播路径中.利用上述原则对该状态转换树重新计算节点复杂度,若常数 C 取值 0.2,那么迭代计算如下:

(1) $V10=1, V11=1, V7=1, V8=1, V12=1$

(2) $V11$ 是叶子节点,$V9$ 算法不变,有

$$V9 = V11/(V11+V12)\times V11 + V12/(V11+V12)\times V12 = 2$$

(3) $V9$ 包含在异常路径中,所以有

$$V9 = C + V9 = 2.2$$

$$V6 = V9/(V9+V10)\times V9 + V10/(V9+V10)\times V10 = 1.825$$

$$V4 = V7 = 1, V5 = V8 = 1$$

(4) $V3$ 未包含在异常路径中,算法不变

$$V3 = V5/(V5+V6)\times V5 + V6/(V5+V6)\times V6 = 1.537$$

$$V2 = V4 = 1$$

(5) $$V1 = V2/(V2+V3)\times V2 + V3/(V2+V3)\times V3 = 1.325$$

对节点进行排名,按照重要度从高到低依次是 $V_q=2$, $V_6=1.825$, $V_3=1.537$, $V_1=1.325$,其余节点重要度均为 1,排名并列.由结果可以看出,节点的重要度发生了变化,包含在异常路径中的节点重要度均有所提升.

4 实验验证

为了验证本文方法的有效性,同时为了方便实验对比,考虑到关键类的确认一般是通过专家直接分析软件系统源代码和阅读理解设计文档来确认的[2].而开源软件 Ant1.6.1 和 JMeter2.0.1 中的关键类已经全部明确:其中 Ant1.6.1 包含 10 个关键类,JMeter2.0.1 包含 14 个关键类.因此,我们选取 2 个规模较大的开源软件 Ant 1.6.1[11]和 JMeter 2.0.1[12]进行实验分析.其中,Ant 是一个常用的 Java 构建工具,代码约 190K 行;JMeter 是一个用于 Web 系统的压力测试平台,可以进行各种情况下的资源占用测试和性能测试,代码约 43K 行.

在实验中分析了 Ant 的核心部分,忽略了外围的扩展.核心部分包含了 664 个类,大约 81K 行代码.选取了 JMeter 的 Core 包进行度量,Core 包内含 277 个类,代码约 38K 行.此外,不同软件系统中类的重要性有差异.因此,类的重要性并不是一个固定的确切数值.对于类重要性度量方法效果的实证验证是通过检测算法能够找到的关键类数目来确定.我们实验软件环境为 Ubuntu 64 位操作系统(版本为 12.04),Open JDK 1.7;硬件环境为 Intel(R) Xeon(R) 3.07 GHz CPU,4G 内存.

具体实验设计为:首先采用的类重要性度量方法(算法 2)进行分析,根据重要性对类进行排名,检查排名前 10%和前 20%的类;接着,我们在考虑异常传播的基础上,使用算法 3 描述的节点复杂度计算方法重新对 Ant1.6.1 和 JMeter2.0.1 中的类进行度量.同样检查前 10%和前 20%的类,实验结果分别如表 1 和表 2 所示.

表 1 Ant 关键类度量对比

KeyClasses	本文算法 2		本文算法 3	
	Examine Top 10%	Examine Top 20%	Examine Top 10%	Examine Top 20%
Project	Success	Success	Success	Success
UnknownElement	Success	Success	Success	Success
Task	Success	Success	Success	Success
Main	Fail	Fail	Fail	Fail
IntrospectionHelper	Success	Success	Success	Success
ProjectHelper	Success	Success	Success	Success
RuntimeConfiguration	Fail	Success	Success	Success
Target	Success	Success	Success	Success
ElementHandler	Fail	Fail	Fail	Fail
TaskContainer	Fail	Fail	Fail	Fail

表 2 JMeter 关键类度量对比

Key Classes	本文算法 2		本文算法 3	
	Examine Top 10%	Examine Top 20%	Examine Top 10%	Examine Top 20%
AbstractAction	Success	Success	Success	Success
JMeterEngine	Fail	Fail	Fail	Fail
JMeterTreeModel	Fail	Success	Success	Success
JMeterThread	Success	Success	Success	Success
JMeterGuiComponent	Fail	Fail	Fail	Fail
PreCompiler	Fail	Fail	Fail	Fail
Sampler	Success	Success	Success	Success
SamplerResult	Fail	Fail	Fail	Fail
TestCompiler	Success	Success	Success	Success
TestElement	Success	Success	Success	Success
TestListener	Fail	Fail	Fail	Success
TestPlan	Fail	Success	Success	Success
TestPlanGui	Fail	Fail	Fail	Fail
ThreadGroup	Success	Success	Success	Success

对 Ant1.6.1 分析的实验中,使用算法 2 检查结果排名前 10%的类能够查找到 10 个关键类中的 6 个,此时召回率为 60%,精度为 9.1%.检查排名前 20%的类能够查找到 7 个关键类,召回率为 70%,精度为 5.3%.使用考虑异常的改进算法时,检查前 10%的类能够查找到 7 个关键类,召回率为 70%,精度为 10.6%.值得注意的是检查前 20%的类并没有发现关键类召回数有

所增加,说明 Main,TaskCon-tainer 和 ElementHandler 这 3 个类的排名始终比较靠后(20%以后).在检查所有的类排名时,发现 Main 类的排名在 300 以后.从设计角度来讲,Ant 作为一个自动化构建工具,主类作为构建入口确实非常重要.但是从程序结构上来讲,Main 类与其余类的交互可能并不是特别多.因此本文方法没有识别出关键类 Main.

对 JMeter2.0.1 的 Core 包分析实验中,使用算法 2,根据检查范围的不同(前 10% 和前 20%),召回率为 42%和 57%,相应的精度为 22%和 14%.使用改进之后的方法检查前 10%和前 20%的类,发现召回率分别为 57%和 64%,相应的精度分别是 29%和 16%.

根据算法 2 和改进算法 3 的实验对比,改进的方法无论在召回率还是精度上均有所提高.说明考虑程序中的异常处理结构可以有效识别程序中的关键类.由表 1 和表 2 可知,扩大检查范围,召回率有所提高,但精度有不同程度下降.

为了进一步说明本文方法的有效性,将本文方法与相关研究工作的成果进行比较.本文中我们选取了 Zaidman(IC_CM 和 IC_CC' + Web 挖掘)[3]、王木生和周毓明等人[1](*h* 指数,*a* 指数)方法进行了对比分析.对比结果如表 3 所示.

表 3　度量方法效率对比

度量方法	Ant1.6.1			JMeter2.0.1		
	召回率(%)	精度(%)	耗时(min)	召回率(%)	精度(%)	耗时(min)
IC_CM	40	21	105	14	7	75
IC_CC' + Web 挖掘	90	47	105.5	93	46	75.5
h 指数	50	5	1	50	6.6	1
a 指数	70	7	1	57	7.5	1
本文算法 2(10%)	60	9.1	10	42	22	4
本文算法 2(20%)	70	5.3	10	57	14	4
本文算法 3(10%)	70	10.6	10	57	29	4
本文算法 3(20%)	70	5.3	10	64	16	4

由表 3 可知,本文方法与 Zaidman 提出的方法相比,精度和召回率以及方法耗时都整体好于 IC_CM 方法.精度和召回率总体来看略有不如 IC_CC' + Web 挖掘的方法,但是运行时间远远小于该方法(约为 10%).在时间效率上有显著的优势.

与基于 *h* 指数的方法相比,本文方法在精度和召回率上均能超越原始的 *h* 指数,与其衍生指数 *a* 指数持平.基于 *h* 指数的方法在 Understand for Java 的分析结果基础上做数据分析,省略了源代码分析的过程,所以其运行时间较短.

本文方法通过静态度量程序中类间交互来识别关键类.该方法的一个预设假定是认为那些与其他类频繁交互的类在对软件功能起到关键作用,从而在程序维护过程中应当作为关键类优先考虑.然而上述假定在实际软件系统中并不总能成立.尤其是面向对象程序包含了多态、继承等特性,使得类之间的关系变得复杂,静态分析不能确定具体的执行情况,需要包含所有的可能性,因此分析过程存在冗余.此外,程序中不可达路径等问题的存在使得某些程序调用不可能发生,而本文方法并没有对此类问题进行处理.上述三个方面的因素均可能影响到本文方法的精度和效率.

5　结论

本文提出了一种基于 UIO 序列的类重要性度量方法,将软件系统抽象为状态转换模型,通过求解转换模型的唯一输入输出序列来构造状态转换树,并通过递归计算状态转换树节点的复杂度来求解类的重要度.针对 Ant 和 JMeter 两个开源 Java 程序的实验表明,与已有的方法相比,本文方法的准确率、召回率以及运行时间均在在可接受的范围内.下一步考虑利用程序动态执行轨迹来记录调用信息,进一步改进本文方法,提高类重要性度量的精度.

参考文献

[1] 王木生,卢红敏,等.利用 h 指数及其衍生度量识别关键类[J].计算机科学与探索,2011,5(10):891 – 903.
Wang Musheng, Lu Hongmin, et al. Identifying key classes using h-index and its variants[J]. Journal of Frontiers of Computer Science and Technology, 2011, 5(10): 891 – 903. (in Chinese)

[2] Zaidman A, Demeyer S. Automatic identification of key classes in a software system using web mining techniques[J]. Journal of Software Maintenance and Evolution: Research and Practice, 2008, 20(6): 387 – 417.

[3] Zaidman A, Calders T, et al. Applying web mining techniques to execution traces to support the program comprehension process[A]. Proceedings of the Conference on Software Maintenance and Reengineering[C]. New York: IEEE, 2005. 134 – 142.

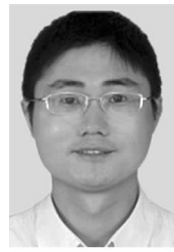
[4] 周欣,陈向葵,等.面向对象系统中基于度量的可复用构件获取机制[J].电子学报,2003,31(5) 649 – 653.
Zhou Xin, Chen Xiangkui, et al. Software measurement based reusable component extraction in object oriented system[J]. Acta Electronica Sinica, 2003, 31(5): 649 – 653. (in Chinese)

[5] 李诺,金茂忠,刘超.一种 Java 程序度量工具的设计实现[J].电子学报,2004,32(S1):175 – 179.
Li Nuo, Jin Maozhong, Liu Chao. A java oriented measuring

tool:Design,implementation and application[J].Acta Electronica Sinica,2004,32(S1):175 – 179.(in Chinese)
[6] Marios F,Nikolaos T,Eleni S,et al.Identification and application of extract class refactorings in object-oriented systems[J].Journal of Systems and Software,2012,85(10):2241 – 2260.
[7] Al Dallal,J.The impact of accounting for special methods in the measurement of object-oriented class cohesion on refactoring and fault prediction activities[J].Journal of Systems and Software,2012,85(5):1042 – 1057.
[8] Sipser M.Introduction to the Theory of Computation [M].Boston,USA:Wadsworth Publishing Co Inc.1997.31 – 47.
[9] Derderian K,Hierons RM,et al.Automated unique input output sequence generation for conformance testing of FSMs[J].The Computer Journal,2006,49(3):331 – 344.
[10] Guo Q,Hierons R,et al.Computing unique input/output sequences using genetic algorithms[J].Formal Approaches to Software Testing,2004,2931:1098 – 1100.
[11] ApacheAnt.The Apache Ant Project[OL].http://ant.apache.org/antlibs/index.html,2014.
[12] Hansen K.Load Testing Your Applications with Apache JMeter[J/OL].Java Boutique Internet,http://javaboutique.internet.com/tutorials/JMeter/,2004.

作者简介

姜淑娟(通信作者) 女,1966 年出生,山东莱阳人,中国矿业大学计算机科学与技术学院教授、博士生导师,CCF 会员,主要研究领域为编译技术,软件工程等.
E-mail:shjjiang@cumt.edu.cn



鞠小林 男,1976 年出生,江苏南通人,南通大学计算机科学与技术学院讲师,中国矿业大学计算机科学与技术学院博士研究生,主要研究领域为软件分析与测试.
E-mail:ju.xl@ntu.edu.cn