软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(6):1368−1381 [doi: 10.3724/SP.J.1001.2012.04072]

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

基于 LoCMD 的软件修改分析技术*

孙小兵[1,2], 李必信[1,2+], 陶传奇[1]

[1](东南大学 计算机科学与工程学院,江苏 南京 211189)

[2](中国科学院 软件研究所 计算机科学国家重点实验室,北京 100190)

Using LoCMD to Support Software Change Analysis

SUN Xiao-Bing[1,2], Li Bi-Xin[1,2+], TAO Chuan-Qi[1]

[1](School of Computer Science and Engineering, Southeast University, Nanjing 211189, China)

[2](State Key Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

+ Corresponding author: E-mail: bx.li@seu.edu.cn

Sun XB, Li BX, Tao CQ. Using LoCMD to support software change analysis. *Journal of Software*, 2012,23(6): 1368−1381. http://www.jos.org.cn/1000-9825/4072.htm

Abstract: Software progression is a fundamental ingredient of software. When changes are made to software, they will inevitably have some unpredicted effects and may cause inconsistencies with other parts of the original software. If the effects induced by the changes affect the whole system, an alternative change proposal may be required instead. Hence, change analysis is necessary before change implementation. This paper presented a compact intermediate representation for object oriented programs based on formal concept analysis—lattice of class and method dependence (LoCMD). Then, based on LoCMD, the study proposes a change analysis model, which includes some activities before change implementation, i.e., program comprehension, impact analysis and change assessment. The empirical study demonstrates the effectiveness of the representation and the change analysis model, and will help maintainers gain a better understanding about the change proposal.

Key words: formal concept analysis (FCA); impact analysis; change analysis; change assessment; lattice of class and method dependence

摘 要: 当对软件进行修改时,肯定会对软件的其他部分造成一些潜在的影响,从而带来软件的不一致性;如果该修改所带来的影响波及到整个系统,可能就需要考虑其他修改方案来实施该修改.因此在实施修改之前,需要对所提出的修改方案进行修改分析,从而确定是否需要进行修改或者选择什么方案进行修改.基于形式概念分析技术,提出了一种紧凑的面向对象程序中间表示——类与方法依赖格(LoCMD);然后,基于 LoCMD,提出了一种修改分析模型,该模型包含了修改实施前一系列软件修改分析活动,包括与修改相关的程序理解、影响分析以及修改评估.实验结果表明了所提出的 LoCMD 和修改模型的有效性,从而有助于维护人员对所提出的修改建议做出正确的理解与决策.

关键词: 形式概念分析;修改影响分析;修改评估;修改分析;类与方法依赖格

* 基金项目: 国家自然科学基金(60973149); 国家教育部博士点基金(20100092110022); 中国科学院软件研究所计算机科学国家重点实验室开放基金(SYSKF1110); 东南大学优秀博士学位论文基金(YBJJ1102)

收稿时间: 2010-07-05; 修改时间: 2011-03-29; 定稿时间: 2011-07-01

中图法分类号: TP311 文献标识码: A

在软件生命周期中,软件维护活动是最困难、成本最高的活动[1].软件的固有特性就是适应性和修改性,软件修改是软件维护的基本元素.修改可以是因为用户提出新的需求、软件使用过程中发现的错误,或者是因为软件所使用的环境发生变化.当对软件进行修改时,肯定会对软件的其他部分造成一些潜在的影响,从而带来软件的不一致性.如果实施该修改所需的成本超过重新开发该软件所需的成本,那么就需要考虑其他修改方案或者重新开发软件.因此在实施修改前,需要对修改进行分析,包括程序理解、影响分析以及对修改评估.修改分析是软件维护的必经阶段.

形式概念分析(formal concept analysis)以数学化的概念和概念层次为人们提供了一种应用数学理论,处理现实世界中形式对象与形式属性之间这种二元关系[2].Ganter 等人将其作为一个较好的数据分析方法,深化、完善了该理论基础,并将它们扩展到各种现实应用中[2].形式概念分析技术在软件工程领域已经得到广泛的应用,如程序理解与分析、逆向工程、重构等[3,4].这些应用的成功主要有两方面的原因: (1) 形式概念分析已经具备了较完善的理论基础以及形式化表示,在应用到软件维护活动中时,更具有形式化方面的表达能力,从而给人以较强的理论和技术说服力;另一方面,形式对象和形式属性这种二元关系经常出现在软件世界中,这种处理二元关系的直觉也推动了形式概念分析在软件领域应用的不断发展.本文将形式概念分析技术应用于修改分析中,主要贡献包括如下几个方面:

1) 构造一种紧凑的适合修改分析的面向对象程序中间表示,类与方法依赖格,LoCMD;
2) 利用 LoCMD 分析软件中程序元素之间的依赖关系,辅助程序理解;
3) 基于 LoCMD 进行影响分析,从类层次的修改集计算方法层次的影响集,并且给影响集中的每种方法赋予一种影响因子度量.根据该影响因子值,可以知道某个方法可能受到影响的可能性;
4) 利用影响分析计算得到的影响集结果进行修改评估,定义一种影响度的度量,该度量表示实施某个修改提议可能对系统产生的影响程度,该影响度可以用来指导维护人员做出是否接受某个修改提议或者选择哪种修改提议的决策;
5) 通过实际中的一个应用程序来验证本文修改分析技术的有效性.从实验结果看,LoCMD 规模比较合理,并且能够有效覆盖类与方法之间的依赖关系;另外,影响分析计算得到的元素影响因子值越大,其受到影响的可能性越大.因而,维护人员可以通过调节影响因子范围来提高影响集结果的精度;最后,通过实际应用程序中的修改提议,利用影响度进行修改评估,该影响度结果与实际中修改所需的成本基本相符.

本文第 1 节陈述一些形式概念分析方面的基本概念.第 2 节给出一种基于形式概念分析的面向对象程序中间表示方法.第 3 节讨论利用形式概念分析来进行修改分析.第 4 节是实验研究.第 5 节是当前修改分析以及形式概念分析在软件工程中应用的一些研究工作.最后一节是结论以及下一步工作.

1 形式概念分析基本概念

形式概念分析提供了一种聚类具有共同形式属性的形式对象的方法,其输入是形式背景,输出是具有偏序(层次)关系的概念格[2].本文为了区分形式概念中的对象、属性与软件中的对象以及属性,将形式概念分析中的对象称为形式对象,而属性称为形式属性.

定义 1(形式背景(formal context)). 通常定义为一个三元组(O,A,R),其中,O 是形式对象集合;A 是形式属性集合;R 是 O 和 A 之间的二元关系,满足 $R\subseteq O\times A$.

给定某个形式背景后,只需满足一定的条件就可以得到该形式背景上的形式概念.

定义 2(形式概念(formal concept)). 形式概念是具有共同形式属性的一组形式对象的最大集合,定义为二元组(X,Y),并且 X 与 Y 之间满足这样的关系:$\tau(A)=X\wedge\sigma(O)=Y$,其中,

$$\tau(A)=\{o\in O|\forall a\in Y:(o,a)\in R\},\sigma(O)=\{a\in A|\forall o\in X:(o,a)\in R\}.$$

X 称为形式概念的外延(extent),而 Y 称为形式概念的内涵(intent).

形式概念之间通常存在着一种层次(偏序)关系,我们通常将这种层次关系定义为概念与子概念的关系.

定义 3(子概念(subconcept)). 一个概念 $co1(X_1,Y_1)$是另一个概念 $co2(X_2,Y_2)$的子概念($co1\leqslant co2$),满足

$$co1\leqslant co2\Leftrightarrow X_1\subseteq X_2\Leftrightarrow Y_2\subseteq Y_1.$$

子概念为构造概念格提供了一种自然的偏序关系.Birkhoff 早在 1940 年就证明了这样的概念格是一种完全格[5],即概念格上任意两个概念均有上确界以及下确界.

定义 4(概念格(concept lattice)). 记为 $L(Co)$,$L(Co)=\{(O,A)\in 2^O\times 2^A|O=X\wedge A=Y\}$.

定义 5(下确界(infimum or join)和上确界(supremum or meet)). 假设有概念 $C1(X_1,Y_1)$和 $C2(X_2,Y_2)$,则:

- 下确界为:$(X_1,Y_1)\wedge(X_2,Y_2)=(X_1\cap X_2,\sigma(X_1\cap X_2))$;
- 上确界为:$(X_1,Y_1)\vee(X_2,Y_2)=(\tau(Y_1\cap Y_2),Y_1\cap Y_2)$.

对于概念格的每个格元素,如果完整地表示它们的外延与内涵,则概念格会变得复杂,不易理解.实际上,存在一种简单的格元素标记方法,使得概念格更容易理解.

定义 6(概念格元素标记(labeling)). 标记为形式属性 $a\in A$ 的元素,其定义为:$\mu(a)=\vee(\{co\in L(Co)|a\in Int[co]\})$;标记为形式对象 $o\in O$ 的元素,其定义为:$\gamma(o)=\wedge(\{co\in L(Co)|o\in Ext[co]\})$.

其中,$Int[co]$和 $Ext[co]$分别是格元素 co 的内涵和外延.当格元素 co 是内涵为 a 的最大的概念时,概念 co 标记为 $a\in A$;当格元素 co 是外延为 o 的最小的概念时,概念 co 标记为 $o\in O$;所有的那些大于$\gamma(o)$的概念在它们的外延中都存在 o,而那些小于$\mu(a)$的概念在他们的内涵中都存在 a.

前面是形式概念分析的一些基本定义,通常,我们在应用该技术时是根据概念格所体现的一些特征来应用的.概念格主要有如下一些特征[2]:

1) 形式概念决定了具有共同形式属性的一组最大的形式对象的集合;
2) 概念格显示了形式对象以及形式属性的层次分类;
3) 概念格中的一些没有用形式对象以及形式属性标记的格元素,可能预示着不同形式对象间存在某种隐性的依赖关系.

概念格是关于形式对象与形式属性之间关系的一个紧凑(compact)并且完整(complete)的表示.概念格上可以挖掘出很多其他可应用的特征,本文关于形式概念分析的应用都是基于上面 3 个特征进行.

2 类与方法依赖格

本文讨论的修改分析是基于源代码的分析进行的,对于源代码的理解和分析通常需要一种抽象的中间表示.这里利用形式概念分析技术来构造面向对象程序的概念格.在讨论对软件进行修改时,通常从设计层的一些抽象模型开始.而在面向对象软件中,从设计层映射到代码层主要是借助于面向对象语言中的类进行.因而,我们讨论的是对哪些类进行修改,然后对其进行修改分析.本文利用形式概念分析技术构造面向对象程序的类与方法依赖格(lattice of class and method dependence,简称 LoCMD).其中,构造 LoCMD 的形式对象是类,形式属性是成员方法;而形式对象与形式属性之间的关系定义为类与方法之间的依赖关系.

定义 7(类与方法依赖关系(class and method dependence)). 类 c 依赖于方法 m,当且仅当满足如下条件之一:

- 方法 m 是类 c 的一个成员;
- 方法 m 是类 c 的任何父类的一个成员;
- 类 c 依赖于方法 k,k 调用 m;
- 类 c 依赖于方法 k,k 被 m 调用.

因此,LoCMD 是一种概念格,其定义如下:

定义 8(类与方法依赖格(LoCMD)). 给定类与方法依赖格 LoCMD,$L(Co)=(N,E)$,C 是由一系列类元素组成的集合,M 是由一系列方法组成的集合,$M(c)$表示类 c 所依赖的方法的集合,有:

$$n\in N\Leftrightarrow Ext[n]=\{c\in C|\forall m\in Int[n]:m\in M(c)\}\wedge Int[n]=\{m\in M|\forall c\in Ext[n]:m\in M(c)\} \quad (1)$$

$$(n,m)\in E \Leftrightarrow Int[n]\subset Int[m]\wedge \nexists k\in N: Int[n]\subset Int[k]\subset Int[m] \quad (2)$$

如上类与方法依赖格定义中,公式(1)表示 LoCMD 上格节点的表示形式,公式(2)表示 LoCMD 上格节点之间的包含关系.

我们通过一个实例来说明我们如何构造面向对象程序的 LoCMD.图 1 是一个简单的实例程序,通过分析,得到类与方法之间的依赖关系,然后构造其形式背景,形式背景即对应于一个二元关系表,可见表 1.

```
class C1{
   private int x;
   public void M1(){x=10;}
   public void M2(){M1();}
}
class C2 extends C1{
   private int y;
   public void M3(){y=5;}
   public int M4(){M1(); }
}
class C3 {
   private C1 c1;
   public void M5 ()
         {c1.M1();}
 }
```

```
class C4 {
   private C2 c2;
   public void M6 (){c2.M3();}
 }
class C5 extends C3{
   private int z;
   public void M7(){M5();}
   public void M8(){z=2; }
}
class C6 {
   private C2 c2;
   private C5 c5;
   public void M9()
         {c2.M3();c5.M8();}
   public void M10(){}
}
```

Fig.1 A simple Java example program

图 1 一个简单的 Java 实例程序

Table 1 Formal context of the example Java program in Fig.1

表 1 图 1 实例程序的形式背景

	*M*1	*M*2	*M*3	*M*4	*M*5	*M*6	*M*7	*M*8	*M*9	*M*10
*C*1	√	√		√	√					
*C*2	√	√	√	√		√			√	
*C*3	√				√		√			
*C*4			√			√				
*C*5					√		√	√	√	
*C*6			√					√	√	√

通过该表,我们使用形式概念分析工具 Galicia(http://www.iro.umontreal.ca/~galicia/)自动生成表 1 形式背景对应的概念格,如图 2 所示.图 2(a)是生成的 LoCMD 的图表示,该概念格的节点分别表示各个节点的形式对象标记(*E* 集合)以及形式属性标记(*I* 集合),通过该概念格,我们可以清晰地看到各个形式概念之间的包含关系以及层次关系.图 2(b)中是每个格元素的完整内容(外延以及内涵),它可以通过概念格上的简单可达性计算得到.

根据 LoCMD 的定义,假设 *C* 是由一系列类元素组成的集合,*M* 是由一系列方法组成的集合,$c\in C$,$M(c)$表示类 *c* 所依赖的方法的集合,我们可以得到如下 3 个引理.

引理 1. LoCMD 上的格元素有且仅有这样两种形式:$(c,M(c))$或者$(C,M(c))$.

引理 2. 如果(C,M)表示某个元素的概念,那么 $M\subseteq M(c)$.

引理 3. 如果概念中内涵的方法个数多于两个,那么这些方法之间存在的关系有且仅有这么两种:它们之间存在传递调用关系;或者它们属于同一个类(或者该类的父类).

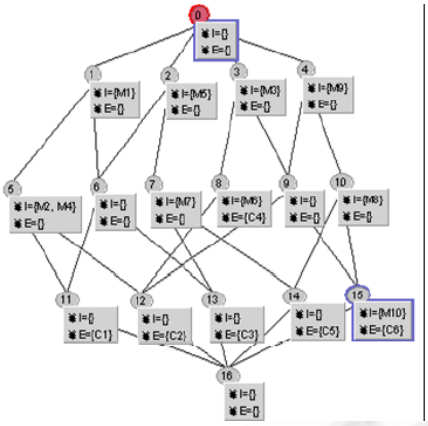
引理说明:

(1) 对于引理 1:如果概念元素中的外延只有单独的某个类 *c*,那么由形式概念的定义,显然,其内涵是由类 *c* 所依赖的方法组成的集合;如果概念元素中的外延由一系列类组成的集合 *C*,那么根据定义 2,该概念元素的内涵是由这些类所依赖的共同方法组成的集合.

(2) 对于引理 2:它是由引理 1 衍生而来,概念元素中的外延是由一系列类组成的集合 *C*,如第(1)条所述,

则内涵是由这些类所依赖的共同方法组成的集合 M,而 $M(c)$是集合 C 中任意一个元素所依赖的方法的集合,由定义 6,则 $M \subseteq M(c)$.

(3) 对于引理 3:根据定义 7,只要满足 4 个条件,我们就称类与方法存在依赖关系,这 4 个条件中的方法可分为两类:一是他们属于同一个类(或者父类);另外一个是他们之间存在传递的调用关系,所以概念元素的内涵中方法之间所存在的关系仅存在这两种.



co0={{C1,C2,C3,C4,C5,C6},{ }}
co1={{C1,C2,C3},{M1}}
co2={{C1,C3,C5},{M5}}
co3={{C2,C4,C6},{M3}}
co4={{C2,C5,C6},{M9}}
co5={{C1,C2},{M1,M2,M4}}
co6={{C1,C3},{M1,M5}}
co7={{C3,C5},{M5,M7}}
co8={{C2,C4},{M3,M6}
co9={{C2,C6},{M3,M9}}
co10={{C5,C6},{M8,M9}}
co11={{C1},{M1,M2,M4,M5}}
co12={{C2},{M1,M2,M3,M4,M6,M9}}
co13={{C3},{M1,M5,M7}}
co14={{C5},{M5,M7,M8,M9}}
co15={{C6},{M3,M8,M9,M10}}
co16={{ },{M1,M2,M3,M4,M5,M6,M7,M8,M9,M10}}

(a) 概念格的图形表示 (b) 列举出每个概念的外延和内涵

Fig.2 Concept lattice for the formal context in Table 1

图 2 表 1 形式背景所生成的概念格

LoCMD 是面向对象程序的一种简单、紧凑的图表示.LoCMD 中元素的形式属性可以通过概念格上的向上可达性计算.如计算 $C6$ 所依赖的方法,首先找到 $C6$ 所对应的格元素($co15$),从该元素向上遍历,可以得到其所依赖的方法{$M10$,$M8$,$M9$}.另外,LoCMD 提供了一种层次化的视图,反映了类与方法的依赖程度,这为影响分析和修改评估提供了依据.

3 基于 LoCMD 的修改分析模型

本文使用形式概念分析技术进行修改分析,如图 3 所示的修改分析模型,首先构造源代码的 LoCMD,然后在 LoCMD 上进行初步的程序理解,再根据这些依赖关系进行影响分析,得到影响集,最后,根据影响集进行修改评估,得到修改评估结果.下面详细介绍修改分析模型中这些活动.

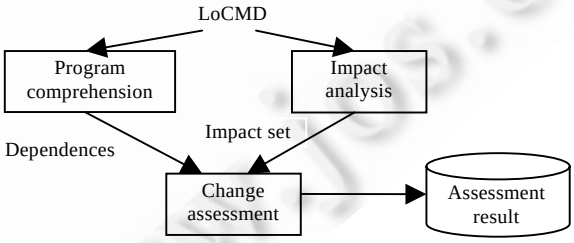


Fig.3 Change analysis model

图 3 修改分析模型

3.1 程序理解

在软件维护过程中,首要工作就是对已有系统进行程序理解,这里主要针对源代码进行理解.前一部分,我

们构造了 LoCMD,通过该概念格,我们可以对程序中类与方法之间的依赖关系进行分析,根据概念格的几个特征,我们可以理解面向对象程序元素之间的一些依赖关系(结合图 2),如下:

(1) 概念格反映了形式对象与形式属性之间的一种层次分类,如图 2 所示,$co10 \leqslant co4(\mu(M8) \leqslant \mu(M9))$,它表示所有依赖于方法 $M8$ 的类也依赖于 $M9$;同样,$co12 \leqslant co8(\gamma(C2) \leqslant \gamma(C4))$,它可以解释为类 $C4$ 所依赖的方法同样也被类 $C2$ 所依赖;

(2) 对应某些类的概念元素 $co14(\gamma(C5))$和 $co15(\gamma(C6))$的上确界所对应的概念元素被方法 $M8$ 所标记,这说明类 $C6$ 和 $C5$ 都依赖于方法 $M8$;类似地,某些方法对应的一些概念元素的下确界是由某个类所标记,这可以解释为该类依赖于所有这些方法;

(3) 由某个类 $C6$ 所对应的概念元素 $co15(\gamma(C6))$向上可达其他一些概念元素,这些概念元素被一些方法所标记,这可以理解为这些方法($M3$,$M8$,$M9$,$M10$)被类 $C6$ 所使用(依赖).

基于以上 3 点解释以及 LoCMD 的 3 个引理,我们可以理解程序中类以及方法之间的依赖关系.在对软件进行修改分析时,对程序中一些元素之间的依赖关系的识别和分析是首要的工作,然后基于这些关系进行后面的影响分析以及修改评估.

3.2 影响分析

当对软件进行修改时,肯定会对软件的其他部分造成一些潜在的影响,从而带来软件的不一致性.软件修改影响分析是用来识别软件修改对软件其他部分所带来的潜在影响[14].影响分析过程的输入是修改集,包含对软件要进行的修改,通过分析,计算得到影响集,它是由受到修改集中元素潜在影响的元素组成的集合.

本文进行影响分析的修改集是由一系列要修改的类组成,而影响集则是受到潜在影响的方法组成的集合,执行修改影响分析的方法主要依赖于 LoCMD 上的两种假设:(1) LoCMD 上向上可达的节点所标记的方法距离要修改的类所标记的格元素越远,其受影响的程度越小;(2) 对于某些要修改的类,在计算某个方法到这些要修改的类的向下可达性时,如果可达的类越多,则其受影响程度越大.这两种假设都是基于概念格的层次特性.对于第(1)个假设,由于 LoCMD 上向上可达的方法被更多的类所依赖,而其中可能存在更多的一些不需要修改的类,所以这些方法所受影响程度更小;第(2)个假设是计算那些要修改的类的上确界,如果某个上确界是更多的要修改的类的上确界,则该上确界所标记的方法受到的影响程度更大.基于这两种假设,我们为 LoCMD 上每一个格元素定义了一种度量,称为影响因子(impact factor,简称 IF),假设某些要修改的类在 LoCMD 上标记的元素为集合$\{co_1,co_2,\ldots,co_n\}$,则 LoCMD 上各元素的影响因子如下定义:

$$IF_j = m + \frac{1}{\sum_{i=1}^{m} \min(dist(i,j)) + 1},$$

其中,m 表示$\{co_1,co_2,\ldots,co_n\}$中能够到达格元素 co_j 所标记的类的个数,$\min(dis(j,i))$表示从格元素 co_i 到 co_j 向上可达所需经过的最少边数.该公式实际上是根据上面两个假设而来,即要修改的类的个数越多,影响因子值越大;如果某个格元素距离要修改的类所标记的格元素越远,其影响因子越小.根据该公式得到的影响集是由一些具有影响因子标记的方法所组成的集合,某方法影响因子越大,则该方法越容易受到这些修改所带来的影响.假设需要修改的类为$\{C_1,C_2,\ldots,C_m\}$,修改影响分析可以按照如下几步进行:

1) 在 LoCMD 上识别这些类所标记的格元素,假设为$\{co_1,co_2,\ldots,co_n\}$;
2) 在 LoCMD 上计算$\{co_1,co_2,\ldots,co_n\}$向上可达的格元素的集合;
3) 根据影响因子度量公式,计算由前两步得到的所有格元素的影响因子;
4) 将上一步中得到的格元素根据他们的影响因子值从大到小进行排序,再计算这些格元素所对应的方法,那么这些方法就是最后受到潜在影响的方法,即为影响集结果.

如图 1 所示程序,假设我们对{$C1$,$C2$,$C6$}这 3 个类进行修改.第 1 步,识别它们在概念格上所标记的格元素,为{$co11$,$co12$,$co15$};第 2 步,计算 LoCMD 上由这 3 个格元素向上可达的格元素的集合,见表 2 的第 2 列(表 2 中只列出了有方法标记的向上可达的格元素);第 3 步,计算这些格元素的影响因子,见表 2 最后一列所示;最后,将

这些格元素排序,以及计算它们所标记的方法,如表 2 中第 3 列所示.表 2 中第 1 列表示权值,某元素权值越高,应该越优先关注.

Table 2 Impact analysis for {*C*1,*C*2,*C*6} changes

表 2 对类{*C*1,*C*2,*C*6}的修改影响分析

Priority	Node	Method	IF
1	*co*5	*M*2, *M*4	2.3
2	*co*1	*M*1	2.2
2	*co*3	*M*3	2.2
2	*co*4	*M*9	2.2
5	*co*15	*M*10	2
6	*co*8	*M*6	1.5
6	*co*10	*M*8	1.5
8	*co*2	*M*5	1.3

给定类层次的修改集,通过影响分析,可以计算得到受到潜在影响的方法,并且这些方法都赋予了一个影响因子度量值,表明该方法可能受到影响的程度,修改评估就是依赖于影响分析的影响集结果进行.

3.3 修改评估

当对软件进行维护时,需要进行修改评估,确定是否实施该修改或者选择相应的修改方案.本文软件修改评估是基于修改影响分析所计算的影响集结果来进行的.首先给定如下修改评估度量,称为影响度(impactness),定义如下:

$$Impactness=\frac{\sum_{i=1}^{m}IF_i}{\sum_{j=1}^{n}IF_j},$$

其中,分母对应的是对程序中所有类进行修改时而带来的影响,n 表示 LoCMD 中所有的类受修改时受影响的方法的个数;而分子则对应的对修改建议的一些类所带来的影响进行预估,m 表示受到修改提议中要修改的类所影响的方法的个数,因而影响度的值总是在 0~1 范围之内.我们可以利用该公式进行修改评估,如果影响度越接近 1,则表示该修改方案所带来的影响程度越大,这样就需要重新考虑修改方案,或者是重新开发整个系统,维护人员可以参考影响度结果来确定是否接受该修改方案,该影响度参考值由维护人员来确定.如果给定多个修改方案,我们可以选择影响度比较小的修改方案作为最终的方案实施修改.

我们还是选择图 1 中的例子程序来说明修改评估,假设现在提出两种修改方案:一是对类{*C*1,*C*2,*C*6}修改,二是对类{*C*2,*C*3,*C*5}修改.我们分别计算这两个方案的影响度,根据影响度公式,分别得到第 1 种方案的影响度结果为 0.72,第 2 种方案的影响度结果为 0.64.从这两个修改方案的影响度结果而可知,第 2 种修改方案更好,对系统带来的潜在影响更小,因而维护人员可以选择第 2 种方案实施修改.

至此,我们完成了软件修改实施前的修改分析活动,通过本文的修改分析模型,可以确定软件是否应该去实施当前修改,或者确定采用哪种方案进行软件维护.如果需要对其进行修改,在我们的修改分析过程中,还有一个重要的副产品——影响集,维护人员可以通过影响集跟踪这次修改带来的波动效应,从而确保软件在这次修改过程中没有引入新的错误或者仍能保持修改后软件的一致性,保证这次修改的成功.

4 实验分析

本节通过一个实际开发的软件系统,从多个角度研究修改分析模型的可行性和有效性.

4.1 实验研究

我们选取的研究实例是实际应用的一个工具 JHSA(Java hierarchical slicing and applications),该工具用于计算 Java 程序的层次切片以及在此基础之上的一些软件维护应用.我们从该工具的版本演化历史库中提取 8 个版本,这 8 个版本的程序统计见表 3.在表 3 中,第 1 行表示 8 个演化版本序号,第 2 行~第 4 行表示这 8 个演化

版本中类的个数、方法的个数以及程序行数.然后,利用这 8 个演化版本去验证 LoCMD 和影响分析技术的有效性以及其修改评估效果.

Table 3 Statistics of the JHSA programs in the 8 evolution versions

表 3 JHSA 程序的 8 个演化版本的统计数据

Version	0	1	2	3	4	5	6	7
Classes	6	7	9	8	12	12	12	12
Methods	101	103	114	111	135	144	145	147
Lines	901	1 193	1 562	1 420	1 821	2 030	2 117	2 243

首先,对于所生成的 LoCMD,我们主要研究其规模以及依赖关系的覆盖程度,即是否覆盖类与方法之间的依赖关系;对于影响分析的有效性,我们主要从精确度和覆盖度这两个度量指标进行有效性验证;对于修改评估能力,我们通过对某个版本提出两种修改方案,然后计算并比较它们的评估结果,再通过手工分析具体的程序来检验其评估能力.

这里,影响分析会用到两个度量指标:精确度(precision)和覆盖度(coverage).精确度用来衡量影响集结果的精确性,表示影响集结果与程序中真正受影响的程序元素的吻合程度;而覆盖度用来度量影响集结果的安全性,表示所计算的影响集结果能够覆盖程序中真正受影响部分的程度.这两个度量指标定义如下:

$$Precision = \frac{|Actual\_Set \cap Estimated\_Set|}{|Estimated\_Set|} \times 100\%,$$

$$Coverage = \frac{|Actual\_Set \cap Estimated\_Set|}{\sum |Actual\_Set|} \times 100\%.$$

在这两个公式中,|*Actual_Set*|表示某个影响因子(IF)范围内所对应的实际影响集元素个数,|*Estimated_Set*|表示某个影响因子范围内使用我们的影响分析技术所计算的受影响元素的个数,$\sum |Actual\_Set|$ 表示实际中所有受影响的元素个数.我们通过版本比较,比较两个版本之间实际发生修改的方法来表示实际的影响集.对于这两个公式,若分母为 0 时,如下处理这种特殊情形:

(1) 若|*Actual_Set*|=0,*Precision*=0,*Coverage*=100%;

(2) 若|*Estimated_Set*|=0,*Precision*=100%,*Coverage*=0.

4.2 实验结果与分析

4.2.1 LoCMD 有效性

我们从两个方面说明 LoCMD 的有效性:一是其规模;二是其覆盖类与方法之间依赖关系的程度,即安全性.表 4 的第 2 行表示的是 JHSA 程序的 8 个演化版本所生成的 LoCMD 的格元素的个数,我们可以看到,这些格元素的个数都在 40 个以下,而传统的依赖图中每一个程序元素(类和类成员)都需要用一个节点表示,因此当一个程序中包含很多类和类成员时,传统的依赖图规模就变得非常庞大;表 4 第 3 行表示的是 JHSA 程序的 8 个版本所对应的概念格元素个数与源程序中类和方法个数总和的比率,从表中可知,LoCMD 的规模只占程序规模(类和方法个数总和)的 20%左右.

Table 4 Statistics of the JHSA programs in the 8 evolution versions

表 4 JHSA 程序的 8 个演化版本的统计数据

Version	0	1	2	3	4	5	6	7
Lattice nodes	18	21	23	27	32	36	38	39
LN/(*C*+*M*)	0.17	0.18	0.19	0.23	0.22	0.23	0.24	0.25

从实验结果来看,LoCMD 的规模比传统的依赖图规模小很多,在 JHSA 程序的 8 个演化版本中,最坏情况下所生成的概念格元素的个数也只占类与方法个数之和的 1/4.另外,对应类与方法之间依赖关系的覆盖程度,我们可以通过覆盖度(coverage)这一度量指标来衡量,因此,这里我们需要考察影响分析计算的所有元素的覆盖度,即 *IF*>1(*IF*>1 时的影响集结果包括了所有可能受影响的方法)时的影响集结果的覆盖度.从图 4(d)中我们可

以看出,当 *IF*>1 时,其覆盖度全是 100%.因此从该结果可以看出,LoCMD 能很好地分析并识别类与方法之间的依赖关系,从而该中间表示是安全的.因此,从规模和安全性两个方面来分析我们的程序中间表示方法,该方法是有效的.

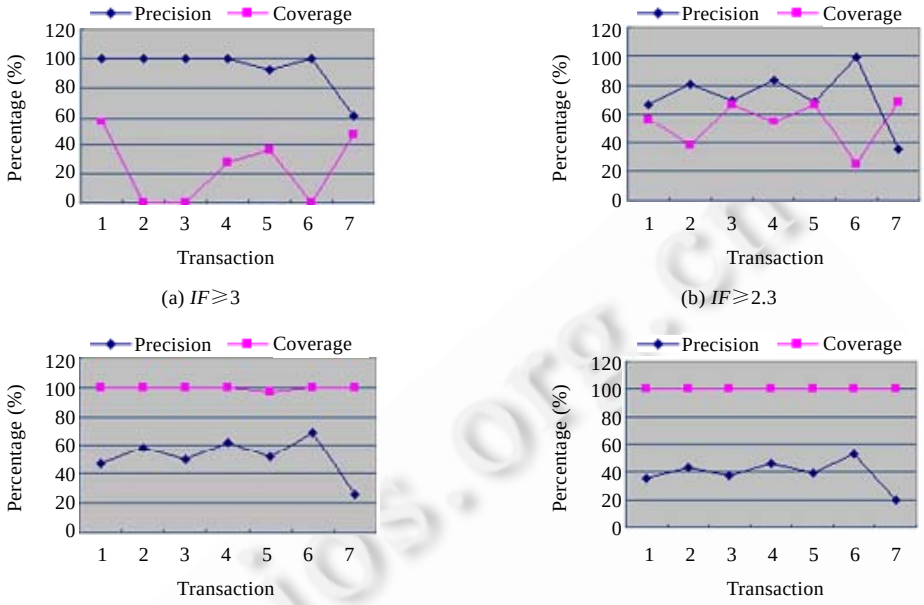


Fig.4 Precision and coverage of estimated methods in different IF ranges

图 4 不同影响因子范围内的影响集结果的精确度与覆盖度

4.2.2 影响分析技术有效性

影响分析技术的有效性主要是从精确度和覆盖度两个方面来考虑.如果覆盖度越高,说明影响分析技术识别了程序中所有可能受影响的元素,那么维护人员对后面的维护活动越有信心;而精确度越高,那么维护人员将会花越少的时间去定位并检查程序中受影响的部分.图 4 表示 JHSA 程序版本演化过程中不同影响因子范围的影响集结果的精确度和覆盖度.图 4(a)是影响因子在 3 以上的方法的精确度和覆盖度折线图,可以看到,大部分方法的精确度都接近 100%.这说明这些方法在实际修改中都需要检查与修改,而其覆盖度比较小,说明该影响因子范围内计算得到的实际受影响的方法比较少,还有很多实际受影响的方法不在该影响因子范围之内.然后,通过图 4(b)~图 4(d)可以看到,随着影响因子值设置的递减,影响集结果中的方法的精确度逐渐变小,而覆盖度逐渐增大.在图 4(d)中可以看到,影响集结果的精确度大多在 40%左右,而覆盖度都是 100%.这说明我们的影响分析技术能够有效地预测所有可能受影响的方法,并且在预测的这些方法中,其精确度可以达到 40%左右,这能够有效地减轻实际修改时维护人员定位程序受影响部分的时间.

4.2.3 修改评估有效性

在本节,我们通过对 JHSA 中一个版本的两种修改方案进行评估,来分析修改评估的有效性.

我们选取 Version 1 作为研究对象,在 Version 1 中一共有 7 个类(*Vertex*,*Edge*,*NodeInfo*,*SDGraph*,*Graph*,*test*,*InnerEdgeIterator*).现在假设有两种修改方案,见表 5 第 1 列,这两种修改方案都是针对 3 个类所作的修改提议.首先,我们生成该版本的 LoCMD,如图 5 所示;然后,根据影响度公式计算这两种修改方案的影响度,见表 5 第 2 列.从表中可知:尽管这两种修改方案都是修改的 3 个类,但是第 1 种修改方案的影响度比较小,只有 0.45;而第 2 种修改方案的影响度高达 0.72,远比第 1 种修改方案的影响度大.所以,可以采用方案 1 作为修改方案.我们通过手工分析源程序,在第 1 种方案对这 3 个类进行修改,只需在 *SDGraph* 中进行简单的一些修改即可;而第 2 种修

改方案中,我们不仅要对 *SDGraph* 中的一些方法进行修改,还要对 *Graph* 中许多方法进行修改.从实际效果来看,采取第 1 种修改方案所带来的影响更小,其维护工作也更轻松.所以采用影响度这个度量进行修改评估是有效的.

Table 5 Two change proposals for Version 1 of JHSA program

表 5 JHSA 程序 Version 1 版本的两种修改方案

Change proposals	Impactness
Vertex, Edge, SDGraph	0.45
Vertex, Edge, NodeInfo	0.72

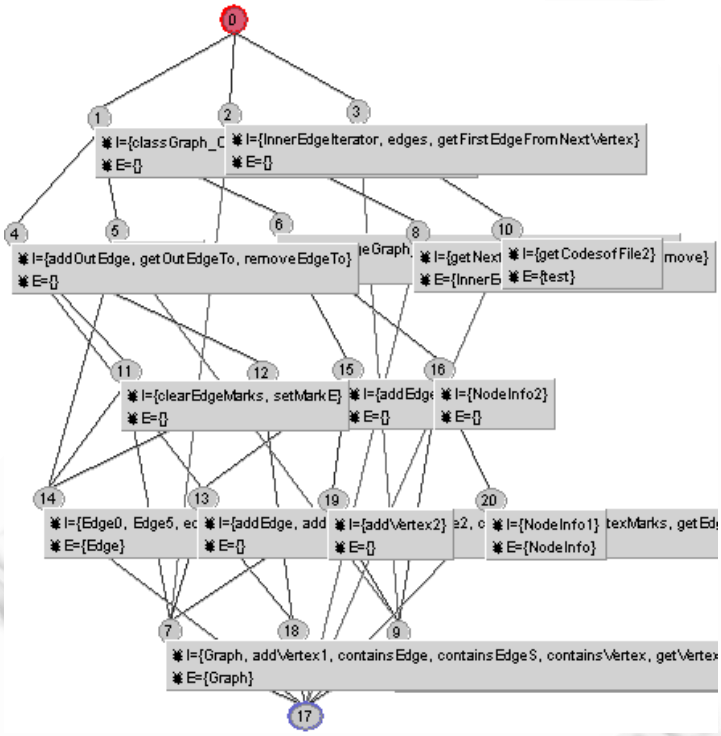


Fig.5 Lattice of class and method dependence for Version 1 of the JHSA program

图 5 JHSA 程序 Version 1 版本的类与方法依赖格

4.3 讨 论

形式概念分析对于修改分析的帮助主要体现在它能够建立一种有效的程序中间表示(类与方法依赖格).尽管该中间表示没有挖掘出更多依赖信息,但是它在修改分析中也存在优势:(1) 该中间表示能够基本覆盖类与方法之间的依赖关系.尽管该中间表示不能区分类与方法之间的依赖关系类型(如继承、调用等),但是它已经能够应付在修改影响分析中的应用,影响分析的思想主要基于对于与修改元素存在依赖关系的元素可能会受到这些修改的影响这样的假设进行的.在传统的中间表示(如系统依赖图)上进行影响分析,在这些中间表示上遍历需要区分不同的边的类型,这就增加了遍历的成本和复杂性,容易出现问题;而在本文的类与方法依赖格上进行遍历,只是一种向上可达性遍历,比较简单;(2) 本文利用概念分析构造的中间表示,从理论上分析,概念格的规模在最坏情况下是 $O(2^n)$.但是出现这种情况时,从另一个方面说明源程序的设计存在很大缺陷,说明程序中各个类之间的耦合性很大.对于这样的代码,对其维护的成本也更大;而实际应用中,它的规模相对于传统的依赖图,其规模更小.传统的依赖图对于程序中的每个类和方法都需要建立一个节点,因而传统的依赖图节点个数至少是程序中类与方法的个数之和;而本文的中间表示,由于类与方法间的依赖关系,尽管可能额外需要一些节点

存储类与方法之间的依赖关系,但是许多类或者方法以及它们之间的依赖关系都存储在一个节点中,因而其规模相对于传统依赖图来说,其规模通常情况下更小,从实验中我们也可以得到这样的结论.表 3 是 JHSA 程序的各个版本的元素统计情况,表 4 是格节点元素的个数以及它们与程序中元素个数的比较.我们可以看到,概念格元素的规模平均只占程序规模(类和方法个数总和)的 20%左右;至于元素之间的依赖关系,图 5 表示的是版本 1 程序生成的类与方法依赖格,该格中节点个数是 21 个,边的个数是 36 条,而版本 1 的程序中类与方法有 110 个,因而至少需要 110 个节点来存储,因而,依赖图的规模会比概念格的规模大.所以,通常情况下,类与方法依赖格相对于传统的程序依赖图,规模会更小.

本文的修改分析模型主要针对在软件维护过程中修改前的 3 个活动:程序理解、影响分析以及修改评估,因此上面的实验分析主要从 3 个方面来说明本文修改分析模型的有效性.尽管我们实验所分析的程序规模还不是很大,但它确实是一个实际开发以及应用的软件.从初步的实验结果来看,我们的方法在影响分析和修改评估中比较简单可行以及有效.另外,我们的方法还产生了一个重要的副产品——影响集,从影响分析这部分实验的有效性来看,我们的影响集结果在影响因子比较大的情况下是比较精确的,能够识别那些真正受修改所带来的波动效应;而在其较小的情况下,又是比较安全的,能够覆盖程序中所有可能受影响的部分.因此在进行修改传播分析时,可以通过调节影响因子的范围去检查那些具有高影响因子值的元素,它们很可能需要实施二次修改.具体的过程是:首先分析那些影响因子比较大的程序元素,对其进行检查与修改,然后再检查影响因子比较小的一些元素,以充分保持修改后程序的一致性.不过,对于影响因子的范围选取,即分析什么范围的影响因子可以充分保证程序的一致性,这个需要反复的实验和验证,我们将在今后的工作中在这方面进行相关的研究.另外,在修改评估中,在什么影响度范围内,我们考虑实施修改或者选取其他修改方案,这也需要实际反复的实验,我们希望在今后的研究中能够提供一些反馈帮助.

5 相关工作

形式概念分析在软件工程中得到广泛关注与成功应用,本文将形式概念分析应用在软件修改实施前的修改分析活动中,利用形式概念分析构造类与方法依赖格;然后,基于该概念格进行程序理解以及修改影响分析;最后,基于影响集结果进行修改评估,修改评估结果可以指导维护人员确定是否接受某修改提议或者采取哪种修改提议.本节主要从两个方面介绍与本文相关的一些研究工作:一是修改分析,二是形式概念分析在软件工程中的一些应用.

5.1 修改分析

软件修改活动是软件维护的关键活动之一,在确定对软件进行维护时,需要对软件进行修改分析,包括程序理解、影响分析以及修改评估.目前,大部分相关的修改分析研究主要集中在修改影响分析以及可修改性评估这两个研究领域.

Tonella 将形式概念分析与程序切片结合起来,用于支持过程内的程序理解以及修改影响分析,他们利用分解切片的概念格作为中间表示进行影响分析;不过,他所提出的方法目前还仅局限于过程内的程序进行分析[9].还有一些学者利用软件修改历史库中存在着大量的程序依赖信息,来挖掘程序的修改信息进行修改影响分析,从而得到与当前修改相关的其他程序元素集合[8,11].孙小兵等人根据不同的修改类型研究它们的影响机制来进行影响分析,这样可以有效地提高影响集结果的精确性[15].另外,Buckner 等人开发了一个将程序理解、影响分析以及修改传播结合在一起的工具 JRipples[10],这是一个交互式工具,可以方便维护人员对修改进行理解以及指导人员进行软件修改.前面这些工作主要是依赖于分析程序的静态信息来进行相关的修改分析,而静态分析技术具有不精确性,一些学者提出了动态修改影响分析的方法.动态影响分析只收集程序实际运行时的一些信息来构造依赖关系,然后通过这些依赖关系进行影响分析,计算影响集,影响集结果规模也比传统的静态分析更小,这样可以有效地减轻维护人员的负担[6,7].以上的一些影响分析方法都是首先构造相关的依赖图,然后在依赖图基础之上进行影响分析,这些方法所需要的成本比较大.在静态影响分析中,所构造的依赖图比较复杂,因而成本比较高,也容易出错;而在动态影响分析中,动态分析需要收集程序运行时的一些相关动态信息,这给影

响分析增加了额外成本;另外,影响分析结果是一个集合,会给维护人员造成不知从何处着手的感觉.本文在修改分析中所使用的影响分析方法,其构造的 LoCMD 规模比较小,在进行影响集计算时只需要向上进行可达性计算,并且最终的影响集是一个赋予优先权值的集合.这样,维护人员可以根据这些值的大小给予优先级检查顺序.

在修改分析领域,还有一些研究人员主要是在修改评估这方面进行研究,主要研究程序的可修改性(changeability).Chaumun 等人也是利用影响分析结果进行可修改性评估,他们针对面向对象程序类之间的不同依赖关系定义各种修改类型的修改影响模型进行影响分析,最后结合影响集结果来分析程序的可修改性,而本文修改评估主要用于度量演化系统的可维护性的一个方面——可修改性[12].另外,Fluri 在代码层次分类不同的修改类型,并定义了不同的可修改性标准,然后根据这些标准建立一种可修改性评估模型[13].该可修改性模型针对某个修改,将程序中各个元素的可修改性分为从高到低的 3 个不同级别,指导维护人员根据该可修改性度量结果去选择相应的修改策略.而本文主要是从软件维护与演化实际出发定义了一个影响度度量,计算某个修改带来的影响程度,用于评估某个修改是否可接受或者选择适当的修改实施方案.

5.2 形式概念分析在软件工程中的应用

近年来,形式概念分析技术在软件工程领域得到广泛的应用,如程序理解与分析、修改影响分析、重构、调试以及测试等[3,4].

形式概念分析在程序理解中的应用最初集中在软件组件检索、配置空间探索、类层次提取以及程序模块评估几个方面[17,28].另外,一些研究人员利用形式概念分析进行特征提取,识别程序中与领域层次修改请求对应的源代码[18,26];还有一些研究工作从代码或者代码演化过程中提取开发过程和演化过程中的模式去理解源程序,这些模式可以是设计模式、可重用的接口模式,也可以是协同修改(co-change)模式,甚至是代码检查的先后模式或者演化模式[19,23].

形式概念分析在修改影响分析方面也有一些研究,主要是过程内的修改影响分析以及类与方法间的影响分析.Tonella 将形式概念分析与程序切片结合起来,用于支持过程内的修改影响分析[9].我们前期也成功地将形式概念分析应用于面向对象程序中的影响分析,主要用于识别方法层次的影响集[25].

形式概念分析很自然地为人们提供了一个层次聚集的视图.这就为识别程序中的层次与聚集打下一个好的基础,目前,将形式概念分析应用重构主要包括类层次结构重构[17]、模块重构[16,29]、由低层次语言规范向高层次的语言规范进行重构[21]等.

软件调试需要执行一系列测试来定位错误,Cellier 等人将关联规则与形式概念分析技术结合进行错误定位[22],而 Ammons 等人给出一种将形式概念分析应用于调试形式化时态规约的方法[20].另外,部分研究人员将形式概念分析技术应用于软件测试方面,主要用于测试用例集生成[24]以及测试用例集约减[27].在测试用例集生成上,Khor 将遗传算法和形式概念分析技术结合起来,旨在生成一个满足分支覆盖标准的测试用例集[24].而 Tallam 等人将形式概念分析技术与贪心算法结合起来进行测试用例集约减.实验结果表明,在满足相同测试需求覆盖能力的前提下,他们的方法在大部分情况下比已有的近似算法可以生成规模更小的测试用例集[27].

本文主要利用形式概念分析构造源程序的中间表示,然后根据该中间表示进行程序理解,影响分析以及修改评估.

6 结束语与展望

软件修改是软件的固有特征,在软件演化过程中,需要对软件进行不断的维护与修改,在每次修改实施前,我们需要对这次修改进行分析,以确定是否实施该修改或者是选择其他修改方案,从而保证软件在演化过程中保持它的一致性和可靠性.本文针对面向对象程序,基于形式概念分析技术,在源代码层次提出一种有效的中间表示——LoCMD;然后,基于该概念格提出一种修改分析模型,该模型中包含了修改分析相关的一些活动,如程序理解、影响分析以及修改评估.通过实验分析验证了我们的修改分析模型的有效性.在未来的研究工作中,我们将把我们的方法应用于实际的软件演化过程中,通过实际应用来确定影响因子的取值范围以保证影响分析

方法的安全性和精确性,以及修改方案决策所对应的影响度取值范围.我们还希望将影响分析产生的影响集应用到修改实施以及修改实施后的一些活动中,从而完整地支持软件维护的各项活动.

References:

[1] Wilde N, Huitt R. Maintenance support for object-oriented programs. IEEE Trans. on Software Engineering, 1992,18(12): 1038−1044. [doi: 10.1109/TSE.1992.1263033]

[2] Ganter B, Wille R. Formal Concept Analysis: Mathematical Foundations. Berlin: Springer-Verlag, 1996.

[3] Tilley T, Cole R, Becker P, Eklund P. A survey of formal concept analysis support for software engineering activities. LNCS 3626: Formal Concept Analysis. Berlin: Springer-Verlag, 2005. 250−271. [doi: 10.1007/11528784_13]

[4] Tonella P. Formal concept analysis in software engineering. In: Proc. of the Int'l Conf. on Software Engineering. Edinburgh, 2004. 743−744. http://portal.acm.org/citation.cfm?id=998675.999497 [doi: 10.1109/ICSE.2004.1317515]

[5] Birkhoff G. Lattice Theory. American Mathematical Society, 1940.

[6] Law J, Rothernel G. Whole program path-based dynamic impact analysis. In: Proc. of the Int'l Conf. on Software Engineering. 2003. 308−318. [doi: 10.1109/ICSE.2003.1201210]

[7] Apiwattanapong T, Orso A, Harrold MJ. Efficient and precise dynamic impact analysis using execute after sequences. In: Proc. of the Int'l Conf. on Software Engineering. 2005. 432−441. [doi: 10.1145/1062455.1062534]

[8] Canfora G, Cerulo L. Fine grained indexing of software repositories to support impact analysis. In: Proc. of the Int'l Workshop on Mining Software Repositories. 2006. 105−111. [doi: 10.1145/1137983.1138009]

[9] Tonella P. Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Trans. on Software Engineering, 2003,29(6):495−509. [doi: 10.1109/TSE.2003.1205178]

[10] Buckner J, Buchta J, Petrenko M, Rajlich V. JRipples: A tool for program comprehension during incremental change. In: Proc. of the Int'l Workshop on Program Comprehension. 2005. 149−152. [doi: 10.1109/WPC.2005.22]

[11] Jashki MA, Zafarani R, Bagheri E. Towards a more efficient static software change impact analysis methods. In: Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2008. 84−90. [doi: 10.1145/1512475.1512493]

[12] Chaumun MA, Kabaili H, Keller RK, Lustman F. A change impact model for changeability assessment in object-oriented software systems. In: Proc. of the 3rd Working Conf. on Software Maintenance and Reengineering. 1999. 130−138. [doi: 10.1109/CSMR.1999.756690]

[13] Fluri B. Assessing changeability by investigating the propagation of change types. In: Proc. of the Int'l Conf. on Software Engineering, Edinburgh (ICSE Companion). 2007. 97−98. [doi: 10.1109/ICSECOMPANION.2007.23]

[14] Bohner S, Arnold R. Software Change Impact Analysis. Los Alamitos: IEEE Computer Society Press, 1996.

[15] Sun XB, Li BX, Tao CQ, Wen WZ, Zhang S. Change impact analysis based on a taxonomy of change types. In: Proc. of the IEEE Int'l Computer Software and Applications Conf. 2010. 373−382. [doi: 10.1109/COMPSAC.2010.45]

[16] Tonella P. Concept analysis for module restructuring. IEEE Trans. on Software Engineering, 2001,27(4):351−363. [doi: 10.1109/32.917524]

[17] Snelting G, Tip F. Reengineering class hierarchies using concept analysis. ACM Trans. on Programming Languages and Systems, 2000,22(3):540−582. [doi: 10.1145/288195.288273]

[18] Eisenbarth T, Koschke R, Simon D. Locating features in source code. IEEE Trans. on Software Engineering, 2003,29(3):195−209. [doi: 10.1109/TSE.2003.1183929]

[19] Tonella P, Antoniol G. Inference of object oriented design patterns. Journal of Software Maintenance, 2001,13(5):309−330. [doi: 10.1002/smr.235]

[20] Ammons G, Mandelin D, Bodik R, Larus JR. Debugging temporal specifications with concept analysis. In: Proc. of the Conf. on Programming Language Design and Implementation. 2003. 182−195. [doi: 10.1145/781131.781152]

[21] Tonella P, Ceccato M. Aspect mining through the formal concept analysis of execution traces. In: Proc. of the Working Conf. on Reverse Engineering. 2004. 112−121. [doi: 10.1109/WCRE.2004.13]

[22] Cellier P. Formal concept analysis applied to fault localization. In: Proc. of the Int'l Conf. on Software Engineering. 2008. 991−994. [doi: 10.1145/1370175.1370220]

[23] Xu JQ, Peng X, Zhao WY. Program clustering for comprehension based on fuzzy formal concept analysis. Journal of Computer Research and Development, 2009,46(9):1556−1566 (in Chinese with English abstract).

[24] Khor S, Grogono P. Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. In: Proc. of the IEEE Int'l Conf. on Automated Software Engineering. 2004. 346−34. [doi: 10.1109/ASE.2004.1342761]

[25] Sun XB, Li BX, Zhang S, Tao CQ. Using lattice of class and method dependence for change impact analysis of object oriented programs. In: Proc. of the 26th Symp. on Applied Computing. 2011. 1444−1449. [doi: 10.1145/1982185.1982495]

[26] Koschke R, Quante J. On dynamic feature location. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. 2005. 86−95. [doi: 10.1145/1101908.1101923]

[27] Tallam S, Gupta N. A concept analysis inspired greedy algorithm for test suite minimization. In: Proc. of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2005. 35−42. [doi: 10.1145/1108768.1108802]

[28] Arévalo G, Ducasse S, Gordillo S, Nierstrasz O. Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. Information and Software Technology, 2010,52(11):1167−1187. [doi: 10.1016/j.infsof.2010.05.010]

[29] Siff M, Reps T. Identifying modules via concept analysis. IEEE Trans. on Software Engineering, 1999,25(6):749−768. [doi: 10.1109/32.824377]

附中文参考文献:

[23] 许佳卿,彭鑫,赵文耘.一种基于模糊形式概念分析的程序聚类方法.计算机研究与发展,2009,46(9):1556−1566.

孙小兵(1985－),男,江苏姜堰人,博士生,主要研究领域为程序分析,软件维护及演化,修改分析.

李必信(1969－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为程序切片技术及其应用,软件演化与维护,软件建模,分析,测试与验证.

陶传奇(1984－),男,博士生,主要研究领域为回归测试,软件维护及演化.