通过用况聚类促进软件结构恢复的方法

任 武
(上海工业自动化仪表研究院,上海 200233)

摘 要: 为促进遗留软件系统的理解和演化任务,需要对系统的整体功能结构有一个总体认识.常见的方法主要是利用聚类技术对静态源代码进行分析.然而在面向对象的软件系统中,由于软件的复杂性如多态和动态绑定等因素的存在,因此需要考虑对象的运行时特性.用况和用况模型从用户的角度描述了软件系统的行为特点,反映了系统的功能特性,已经成为程序理解的一个关键所在.通过对用况的聚类分析,建立软件系统的结构框架,并与实际的设计结构进行比较,以验证软件结构质量好坏,是本文所提出的研究思路.该方法结合利用关联规则挖掘技术获取用况对应的执行事件,并通过静态结构的分析对用况和相关事件的调用依赖关系进行扩充.最后通过一个开源项目的实验分析进行评估,讨论方法的适用性和有效性.

关键词: 动态分析; 静态分析; 用况; 聚类; 软件结构恢复
中图分类号: TP311.5 **文献标识码:** A **文章编号:** 0372-2112 (2013)07-1412-07
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2013.07.026

Approach for Facilitating Structural Recovery by Clustering Use Cases

REN Wu
(*Shanghai Institute of Process Automation Instrumentation*, *Shanghai* 200233, *China*)

Abstract: In order to promote the understanding and evolution of a legacy software system, it is necessary to have an overall view of the structure. The common solution is to analyze the static source code by using clustering technique. In object-oriented software systems, however, due to the complexity of the software itself, such as exists of polymorphism and dynamic binding, and other factors, object's behavior actives need to be considered also. Use cases reflect the functionality of a system and play a key role in program comprehension. By clustering to establish the structural framework of the system, and by comparison with the actual design of the structure, our approach combines association rule mining techniques to obtain the mapping between use cases and corresponding events, and through the analysis of the static structure of use cases and related events invoked dependencies expansion. Finally, the experimental analysis of an open source project is evaluated and discussed to account for the applicability and effectiveness of our approach.

Key words: dynamic analysis; static analysis; use case; clustering; software structural recovery

1 引言

软件架构是进行软件设计、维护和程序理解的一个重要制品.当前的软件维护工作的一个主要目标是恢复遗留系统的结构模型及结构信息,包括功能组件的结构信息和业务规则.然而在维护过程中,一个现实问题是遗留系统往往缺乏详细、规范的系统需求和设计文档,或者由于随着软件版本的演化,相关文档缺乏必要的更新.在这种情况下,维护人员通常采用软件聚类的方法从源代码中提取相关的特征组件结构,即对代码元素按照功能组件聚集成组,将系统分解成更小的,更易于管理的簇[1],这是提取高层结构视图的一项重要技术.

现有的聚类分析技术应用在软件系统中,通常利用静态的软件结构信息,如从代码中提取包含继承、依赖和类型引用等组件之间的关系进行聚类,这种方法基于有限的静态源代码分析,不能分析系统组件之间的动态交互特性,容易将与特征无关的组件进行聚类.而基于文本词汇的信息如文件名称和代码注释等进行聚类的方法,在技术上也有明显的缺点,比如需要依靠现有的设计规范,数据中存在噪声和模糊性等,因而从文本词汇这种非正式的信息源中提取知识往往是不可靠的[2].

对于面向对象的系统来说,聚类是按照对象相似度将相类似的对象聚合在一起的过程.聚类的结果是不同的簇群,使得同一簇中的对象比在不同簇中对象有更高

收稿日期:2011-10-22;修回日期:2013-03-06;责任编辑:蓝红杰

的相似性[3].传统的聚类技术在进行架构分解时通常依靠代码的静态分析.而近期的研究表明,基于动态分析的聚类能更好的产生有意义的系统结构视图.通过收集执行轨迹和动态分析,可以容易确定出,在一个系统中软件实体的功能表现,以及在实现特定功能中代码实体间的交互关系.本文提出的聚类方法,首先建立在这种行为特征方面而不是仅仅依靠单纯的静态结构关系.

执行用况场景来表现系统功能需求是最常用的方法.用况从宏观的角度上描述了软件系统的外部行为特征,因此遗留系统的用况模型恢复,已经成为在较高的抽象层次上进行逆向工程和程序理解的一个关键技术.本文基于这种思想,引入了动态代码分析技术,通过收集事件轨迹中方法调用集合,并利用数据挖掘的手段,逆向恢复出软件系统的结构模型,以评估系统的结构质量,进而帮助程序理解和可能的系统重构任务.

本文提出了通过聚类恢复系统的结构的方法.在设置场景和用况的对应时,假设是多对多的映射关系.这摆脱了早期研究中通常认为的一对一映射关系.这样做一方面符合用况是对多个场景抽象的情况,另一方面也减轻了维护人员在设置用况和场景时遇到的困扰.例如对于"验证权限"用况,对应的场景可分别为"1.User enters Guest and password;2.System validates login and password;3.System opens a session".[9]在建立用况和调用模块对应时,我们采用了基于强关联规则进行过滤的方法,以挖掘出用况和模块间的潜在关联.

本文的聚类分析结合了动态跟踪和静态模块依赖分析两方面的方法.这部分已经有大量的相关工作:在软件聚类方面,前期的研究主要集中在静态依赖性的分析,计算代码对象间的相似度.后期的研究也包括采用动态分析进行聚类的.文献[1]介绍了基于静态组件依赖图的聚类方法,采用了通过对组件间相互调用的次数取权重的动态分析方法,但并没有利用动态分析建立组件间的依赖关系.文献[4]提出了通过动态和静态结合的特征定位方法.使用概念分析技术建立组件和特征的关联.但建立起的关联仍然需要大量的人工参与,以分辨相关联的程度.而本文通过挖掘关联规则的方法对用况和组件构建关联矩阵,从而实现了明确的关联映射.文献[2]提出了动静态结合的软件聚类方法,基本思想是使用软件特征作为聚类的标准,这方面和本文采用的基于用况作为聚类的方法相似.但前者聚类的前提是每个场景对应一个特征,每个特征对应一条轨迹,这对场景初始设置要求比较高,且没有进行多次循环迭代设置场景的考虑.

2　基于用况聚类的方法

本文的方法结合两种不同的信息来源,分别是执行场景获取的动态事件和对每个事件通过静态分析获取的语法信息.每个事件所对应的实体经过模块依赖图和抽象语法树分析,提取出元数据如组件依赖性以及方法的调用关系等存储在事实库中,根据检索的对应事件补充到用况—组件关系矩阵中.

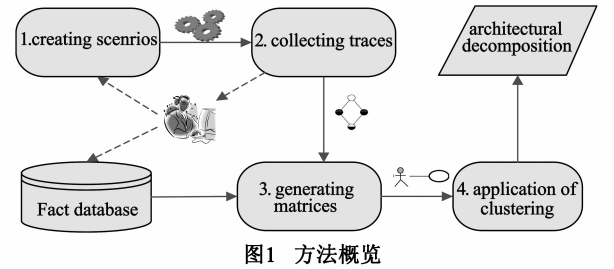

图1　方法概览

(1)场景设计　用况的一个场景是指用况执行过程中的一个特定的实例或执行路径.场景是用户输入一系列的触发动作产生可观察的结果[4].在场景运行时收集一系列发生的跟踪事件,每个事件可以对应到不同粒度级别的调用实体如方法或类.一条执行轨迹是由描述特征动态行为的运行时的事件序列组成.通过执行阶段用况建立起场景和执行实体间的对应关系,每个场景是目标系统的一次执行过程,其中包含了一个或多个系统用况.每个场景选择尽可能功能相关且特定的系统用况,这样做利于收集到较准确的对应于特定用况的执行事件.

(2)动态分析　首先运行一系列与各种的特征相关的场景,并使用相关的代码植入工具和策略来收集系统执行的动态信息,如目标运行时的方法调用序列,动态分析提取执行事件的运行时行为特性.本文采用Eclipse测试和性能平台(TPTP)插装代码和收集执行轨迹.TPTP是一个开放平台,提供应用监控,追踪和profiling等的服务.通过对插装的软件系统运行场景,以entry/exit调用列表的形式,获取每个场景的执行轨迹,所有的场景,场景中包含的用况集合以及方法调用序列组成了一个二维关系矩阵,$\boldsymbol{R}_{su} \subset S \times U$ 和 $\boldsymbol{R}_{sm} \subset S \times M$ 基于两种调用关系,其中 S 表示一系列场景,U 表示包含的用况,M 表示方法调用.表1显示了这种关联的一个例子,表中"×"表示存在调用关系.

表1　调用关系的例子

	*U*1	*U*2	*U*3	*M*1	*M*2	*M*3	*M*4	*M*5	*M*6	*M*7
*S*1	×		×	×			×		×	
*S*2	×	×			×		×	×		×
*S*3		×	×			×	×	×	×	×
*S*4	×			×	×			×		×
*S*5		×	×	×			×			×

定义1　方法调用与包含的特定用况的关系:$\forall s \in S\{s_i\}, \forall m \in M\{M_j\}, \exists\{m \mid R_{sm}(s,m)\}$,对于一个特

定的用况 u_i,所有与其无关的方法调用可以表示为:$m \notin \{m \mid R_{sm}(s,m) \wedge \neg R_{su}(s,u_i)\}$

(3)关联规则挖掘 在程序执行中,触发不同的用况或场景,通过多重执行产生不同的轨迹.这些执行事件如从系统不同部分获取的方法,可能在实现多重用况中出现多次.所以,属于一个类的方法能够出现在多个用况轨迹中.这样有可能在执行轨迹中,识别重复出现的模式.以便发现方法执行中最相关的关联性.一旦收集了执行轨迹,采用工具运行数据挖掘算法,通过关联规则学习,发现与每个用况高度相关的事件的定位.

定义 2 关联规则:形式上,关联规则项目集 I 包含 N 个二进制属性,$I=\{i_1,i_2,i_3,\cdots,i_n\}$,事务集 D 包含 M 个事务,$D=\{t_1,t_2,t_3,\cdots,t_m\}$,$D$ 中的每个事务包含 I 项目集的一个子集.规则定义为一个 $X \Rightarrow Y$ 实现,其中,$X,Y \subseteq I$ 且 $X \cap Y=\varnothing$.

在表 1 例子中,项目集是所有方法的单元,在执行轨迹和用况中,事务作为执行场景.在关联规则学习中,置信度和支持度是常用的两个度量标准.

定义 3 支持度和置信度:对于形如 $X \Rightarrow Y$ 的关联规则,规则的支持度和置信度表示为

$$SUPPORT(X \Rightarrow Y)=P(X \cup Y)$$

$$CONFIDENCE(X \Rightarrow Y)=P(Y \mid X)$$

对一个方法 M 考虑作为与用况 U 相关的定位,在方法和用况间的规则的置信度值用于确定排列是否高度相关.在表 1 例子中,支持度 $(U1 \Rightarrow M1)=2/5=0.4$;置信度 $(U1 \Rightarrow M1)=2/3=0.67$;如果设置最小置信度为 0.5,满足强关联规则的条件,表明用况 $U1$ 的执行暗示方法 $M1$ 的存在,即在用况 $U1$ 中包含调用方法 $M1$ 的概率.

按照上述的方法,对表 1 的事务集设置最小置信度 $(U \Rightarrow M)$为 0.3,最小支持度 $(U \Rightarrow M)$为 0.3,挖掘频繁 2-项集后得到如下强关联规则:

$\{[m7,u3]=0.4,[u3,m1]=0.4,[u1,m4]=0.4,[u1,m1]=0.4,[u1,m2]=0.4,[u1,m7]=0.4,[m6,u3]=0.4,[u2,m7]=0.6,[u2,m4]=0.6,[m5,u1]=0.4,[u2,m5]=0.4,[u3,m4]=0.6\}$

我们以用况和关联挖掘得到的规则组成一个关系矩阵,如表 2 所示,通过设置最小置信度和支持度,可以从复杂的数据中获取用况和调用方法间直接的关联.

表 2 转换后的调用关系表

	M1	M2	M3	M4	M5	M6	M7
U1	×	×		×	×		×
U2				×	×		×
U3	×			×		×	×

定义 4 形式化概念格(Formal Concept):给定一个对象集合 $X \subseteq O$,X 的公共属性 $A(X)$被定义为:$A(X)=\{a \in A \mid (o,a) \in R \wedge \forall o \in X\}$属性集合 $Y \subseteq A$ 的公共对象 $O(Y)$被定义为:$O(Y)=\{o \in O \mid (o,a) \in R \wedge \forall a \in Y\}$.

二元组 $c=(X,Y)$被称为一个概念,当且仅当 $Y=O(X)$并且 $X=A(Y)$. $X \subseteq O$ 被称为概念的外延(extent),$Y \subseteq A$ 被称为概念的内涵(intent)[5].

根据定义 4,图 3 中 $u2$ 对应的概念{(u2, u1),(m4, m5, m7)},$u1$ 对应的概念{(u),(m1, m2, m4, m5, m7)},$u3$ 对应的概念{(u3),(m1, m4, m6, m7)},$m3$ 对应的概念{(∅),(m1, m2, m3, m4, m5, m6, m7)},$m4$ 和 $m7$ 对应的概念{(u1, u2, u3),(∅)}.从图 2 中可见 $u2$ 连接到 $m4$ 和 $m7$,但从图形上无法确定连接 $m5$,因此会造成关系的遗漏.通过提取强关联规则,可以在图 3 中清晰的表现出各个用况与调用方法间的关联.

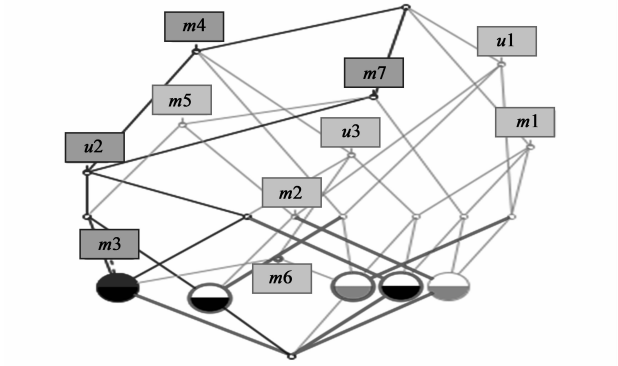

图2 概念格表示

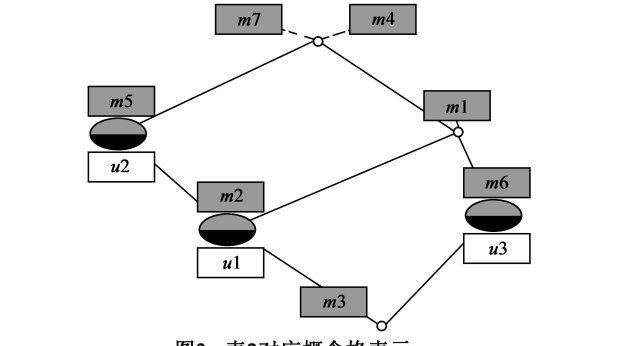

图3 表2对应概念格表示

(4)静态依赖图提取 模块依赖图用于认定代码模块之间的各种调用依赖关系.代码模块可以是类或方法作为粒度.例如,在方法粒度上,初始分析的输出是一个图形,所有方法和变量是节点,节点间关系作为有向边.

定义 5 调用依赖性:设用况 U 和模块集 M 构成二元关系,$R1 \subseteq U \times M$,$\forall m \in u,(u_i,m) \in R1$. $R2 \subseteq U$

$\times(M\times M),\forall p,q\in u\&\&p\rightarrow q,(u_i,p,q)\in R2$

定义 6　静态类之间依赖关系:设 $C(S)$是系统的类集合,$R(C)$ 表示类的依赖关系,结构类的依赖性包括类的继承、聚合和类的方法调用等,则结构类的依赖性定义如下:

$$\forall c_p,c_q\in C(S),(c_p,c_q)\in R(C)\Leftrightarrow$$
$$c_p\{\text{uses}\vee\text{aggregates}\vee\text{composites}\vee\text{extends}\}c_q$$

根据定义 5,扩展动态调用的模块到静态程序结构中,具体到每个调用模块,根据定义 6 进行补充.在分析静态依赖关系中,不是分析所有的模块类型,而是只考虑在用况中出现的事件的静态类结构依赖.同样,在对选择的用况考虑时,也进行相应的扩展分析.在此基础上应用聚类算法,从而建立起系统的结构模型.

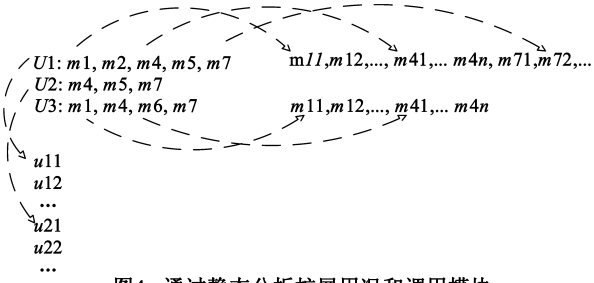

图4　通过静态分析扩展用况和调用模块

图 4 显示了通过静态分析补充用况和调用模块的方法.比如,在用况 $u1$ 对应的事件集中,包括 $m1$,$m2$,$m4$,$m5$,$m7$ 各个方法,针对各个方法,进行静态代码的依赖性分析,根据各模块的依赖图,确定方法间的调用依赖关系,并补充到用况对应的轨迹事件中.同时,也考虑对用况的扩展,根据各个用况在源代码中的分支语句进行用况提取.通过对分支信息的分析以及剪枝算法的运用得到调用轨迹的划分,从而获取候选用况.我们提出用况扩展算法,如算法 1.

算法 1　用况扩展算法伪代码

```
输入:用况事件轨迹与 AST
输出:候选用况子集
FOR EACH 用况执行轨迹中的事件 DO
    检索事件对应 AST 代码中的条件语句
FOR EACH 条件语句对应的节点 DO
  IF 查找的模块 ∉ SUB(当前节点)THEN
    从当前节点回溯
    检查是否有语句满足选择条件
    查找对应的子集,替换对应的子节点
    将候选节点并入子集中
ELSE
    继续寻找下一个节点
ENDIF
    返回候选用况相关的子集
```

(5)聚类分析　在形成用况—调用模块的关系矩阵后,利用聚类算法,分别对用况和模块进行聚类形成不同的视图.聚类有基于划分的聚类和层次聚类的不同算法,基于划分的聚类算法需要设置给定聚类的数量,而层次聚类,比如凝聚层次聚类,是自底而上形成系统树图后,通过切割值确定系统的聚类.本文的案例实验采用的是后一种聚类方法.

(6)实验评估　为检验聚类的质量,我们将实验获取的聚类结果同实验的真实系统结构进行比较评估.为此,采用传统的算法 MoJoFM 距离计算两者之间的相似程度[6].MoJoFM 算法用于计算从一个划分到另一个划分变换所需的"移动"和"合并"操作的最少步数.给定两个划分 A 和 B,则存在如下的"质量度量"计算公式:

$$MoJoFM=\left(1-\frac{mno(A,B)}{\max(mno(\forall A,B)}\right)\times 100\%$$

$\max(mno(\forall A,B)$表示在最坏的情况下,任意的划分 A 到划分 B 的转换所需要的操作数.MoJoFM 的取值范围在 0% 到 100% 区间,表示两个划分从完全不同到完全相等,数值越大,表示越相似.数值的计算通过 MoJoFM 工具实现.为了计算两个划分的距离,需要对实验的对象提取依赖关系,保存在 TA 或 RSF 格式文件中作为工具的输入.文件格式要求包括软件系统的依赖关系名称的提取,两个比较的文件应当具有相同的对象集合等.

3　案例研究

本节使用开源项目 JHotDraw V5.2 作为实验对象进一步说明本文方法的流程及实现细节,并将实验结果同目标系统的实际设计进行比较.

JHotDraw[7]系统是开源系统,描述 2D 图形的 JAVA 框架.其中 JHotDraw 5.2 包含 8 个包,156 个类.图 5 显示了系统的包结构,每个包有不同数量的类:其中 util

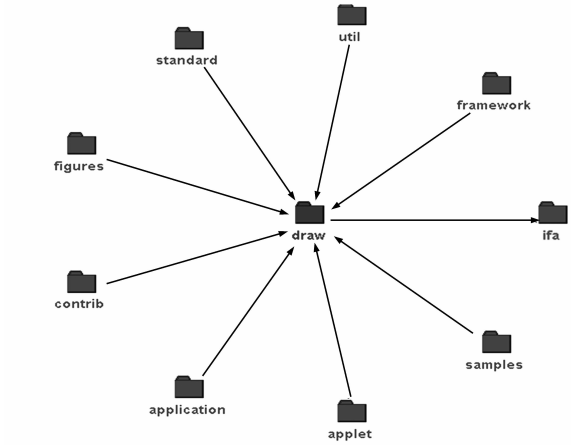

图5　JHotDraw V5.2 包结构

有 24 个类,standard 有 55 个类,figures 有 34 个类,application 有 1 个类,applet 有 2 个类,contrib 有 18 个类,samples 有 4 个类,framework 包含 18 个类.

表 3 显示了实验所选择的用况.实验中选择尽可能多的不同方面的用况,以便覆盖系统各部分代码,取得特征分布的平衡[8].针对每个用况设置不同的场景,如表中用况 1,我们设置场景 1:select the rectangle figure element;场景 2:select the round rectangle button;场景 3:select the ellipse button to draw a new ellipse figure 等.触发这些场景,每个场景生成一个跟踪轨迹,这样收集到类似表 1 的场景-用况和调用模块的事件.然后运用上节所述的挖掘关联规则的方法,得到如表 4 的用况-调用模块的关联矩阵,这里选择类作为调用模块的粒度.

表 3 实验初期选择的用况

1	for each graphical element in the load file:select it
2	for each graphical element in the load file:select it and move it
3	for each graphical element in the load file:turn tracing off select it turn tracing on resize it
4	crate a new text field using the text button
5	use the attach text button to attach a text to each existing graphical element
6	use the URL button to attach an URL to each graphical element
7	use the line button to draw a new line figure
8	use the connector button to create a connector between each graphical element,stating from the rectangle clockwise draw all the arrow not already present
9	using the scribble tool draw a line made by three segment,a smooth part and a final segment
10	use the border button to add a browser to each figure,starting from the rectangle clockwise
11	for each graphical element in the load file staring from the connectors:select it use copy command from the application menu
12	maximize the window,for each graphical element in the load file:select it use text-color property from the menu and select red

对每个用况,我们调查跟踪事件中的静态依赖关系,获得方法类之间的调用作为每个用况的事件的补充.比如:DrawApplication-> CommandMenu;AbstractFigure -> FigureChangeEvent;StandardDrawingView -> FigureEnumerator;DecoratorFigure:displayBox -> RectangleFigure:displayBox; CreationTool: mouseUp -> DrawApplication: toolDone; StandardDrawingView -> AbstractTool; UngroupCommand: isExecutable -> StandardDrawingView: selectionCount…

根据上节提出的结合静态软件结构的分析的算法考虑用况的扩展.方法对目标程序进行分析的时候考虑程序重点分支语句,分支语句(如 if 语句、case 语句和 while 循环语句)被认为是用况与用况之间界定的标志之一.对于包含出口函数的分支语句而言,在基于线索的用况模型恢复方法中界定为不同用况,结合静态的组件结构关系和整合多个场景用况图 UCM(Use Case Maps)来反映系统行为模型.候选用况将反馈给分析人员来裁剪,以补充结构恢复的完整性.如图 6 所示.

表 4 转换后部分用况-调用模块的关系表

1	StandardDrawing, FigureChangeEventMulticaster, AttributeFigure, EllipseFigure, FigureChangeEvent, SendToBackCommand, BringToFrontCommand, FigureAttributes, CompositeFigure, StandardDrawingView, DecoratorFigure, AnimationDecorator, SimpleUpdateStrategy, ColorMap
2	MySelectionTool, PaletteButton, SendToBackCommand, BringToFrontCommand, AttributeFigure, DragTracker, StandardDrawingView, RectangleFigure, ToolButton, DecoratorFigure, FigureEnumerator, FigureChangeEvent, ReverseVectorEnumerator, RelativeLocatorBoxHandleKit…
3	StandardDrawing, FigureChangeEventMulticaster, CutCommand, HandleTracker, MySelectionTool, PaletteButton, BringToFrontCommand, AttributeFigure, ReverseFigureEnumerator, StandardDrawingView, DrawApplication, AbstractHandle, FigureChangeEvent, CompositeFigure…
4	CreationTool, TextFigure, CompositeFigure, CompositeFigure, DecoratorFigure, ReverseFigureEnumerator, CutCommand, PaletteButton, DrawApplication, AbstractTool, CompositeFigure, ReverseFigureEnumerator, DuplicateCommand, SimpleUpdateStrategy, DrawingChangeEvent…
.	…

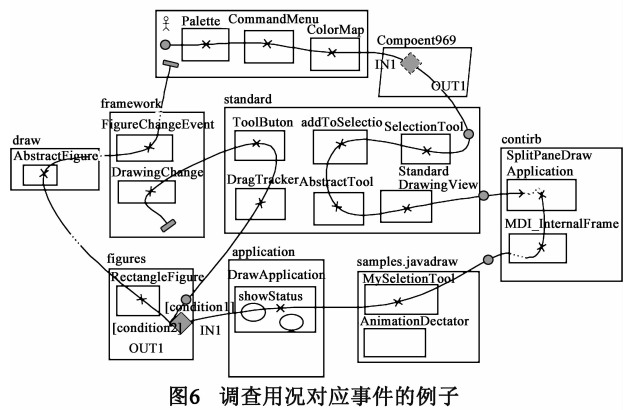

图6 调查用况对应事件的例子

在图 6 中,以用况 1 的事件作为例子来说明本文所采用的扩展方法.在收集到用况 1 对应的事件后,通过 UCM 将用况对应的事件以包和类的粒度来表现出来.同时,调查每个对应事件的静态调用关系,如果在动态跟踪事件上没有出现,则补充到用况事件中.如包 standard 的 StandardDrawingView 类,依赖 contrib 包中的 MDI_InternalFrame 类和 SplitPane_DrawApplication 类,通过分析补充到用况 1 对应的事件中.同时,根据静态结构的条件分析,界定不同的用况,并根据新增的用况,重新设置执行场景,收集该用况对应的调用事件,因此,采用的方法是迭代循环的,根据逆向恢复需求的变化可以随时调整场景和迭代次数,增加流程的可控制性和适应性,以此提高系统恢复的完整性.接下来对获取的

矩阵进行层次聚类处理,得到如图 7 所示结果,树状的结构代表在算法不同阶段形成的簇.通过对系统树设置切割点确定系统的聚类集.具体说,实验中设置切割值得到如表 5 所示的聚类集.

我们将实验产生的聚类进行处理以 RSF 格式保存后,输入到 MoJoFM 工具中,通过设置双向计算,该工具运行计算出移动和合并的度量指标.从实验数据可以看出,方法结果接近于 JHotDraw 5.2 设计者提供的结构分解.

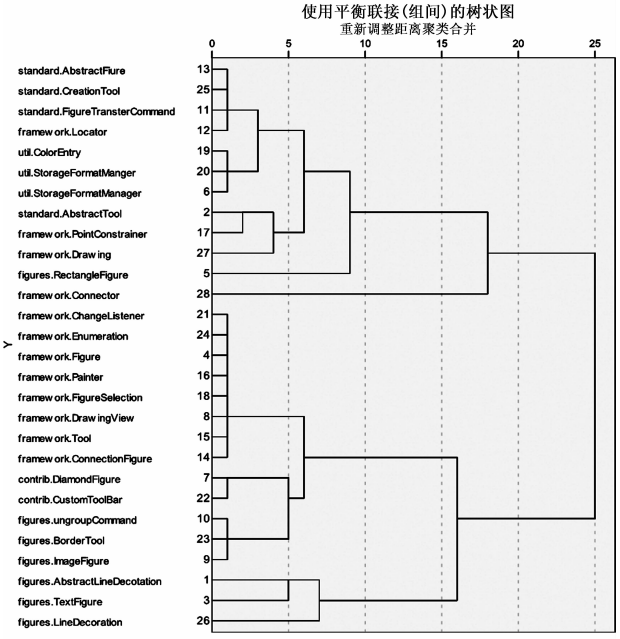

图7　应用聚类算法后产生的系统树图

表 5　切割系统树图后得到的类

cluster	classes invoked in the traces
C1	standard. AbstractTool, framework. PointConstrainer, framework. Drawing
C2	figures. RectangleFigure
C3	standard. AbstractFigure, standard. CreationTool, standard. Figure-TransterCommand, framework. Locator, util. colorEntry, util. StorageFormatManger
C4	framework. ChangeListener, framework. Figure, framework. Painter, framework. FigureSelection, framework. DrawingView, framework. Tool
C5	contrib. DiamondFigure, contrib. CustomToolBar, figure. UngroupCommand
C6	figures. AbstractLineDectation, figures. TextFigure, figues. LineDecoration

显然,MoJoFM 的结果只是反映出一个度量数字,代表聚类的质量.为了进一步分析实验的结果,还需要将形成的聚类同实际的包结构进行比较,以便找出差异.从表 5 的实验数据可以看出,C3 对应于 standard 包,C4 对应于 framework 包,C5 有两个类是 contrib 包,一个是 figure 包,C6 聚类的类都是 figure 包,C1 的聚类结果较差.从实验也可看出,层次聚类的切割值对实验的结果有较大的影响,这需要经验判断.为了比较本实验方法与实际的软件结构划分方法,按照参考划分的数量处理聚类结果.这样有可能造成分布的不均衡,如 C2 聚类结果只有 1 个类,出现所谓的星云现象[3].而其他分布较为均衡.我们考虑有这么几个因素:首先选择的实验对象的粒度较大,是类层次级别,JHotDraw5.2 实际的类为 156 个,其中还包括 util 包中公共函数.其次,在实验过程中,通过执行用况场景来获取事件,虽然通过用况和静态结构分析来扩充分析的范围,但无疑实验的结果与初期设置的用况密切相关.针对架构的恢复,如果只选择代表相类似功能的特征用况,结果恢复出的结构,很可能以包含大部分实体的一小部分簇而告终,即所谓的黑洞现象[3].所以如何选择覆盖系统各部分的特征是需要进行迭代和比较的.本文的方法也试图通过扩充用况来取得一个平衡集.在分析方式上,具有场景的动态分析基本上是面向用况或特征的聚类方式,这样就容易形成系统的功能结构框架,也就是说是需求特征层次的结构恢复.这与基于静态代码的聚类或面向模式的聚类方式相比,在恢复出的结构模型上也许存在着一些差异.

4　总结和展望

本文的主要思想是提出了对执行用况场景的事件的聚类达到结构恢复的方法.通过逐步扩展用况的分析对象,帮助系统模型的整体恢复,并利用对静态程序依赖和语句中的分支条件的分析结果,产生调用方法和用况,作为知识被反馈到用况场景设计步骤中.循环迭代的目的是补充新的特征相关的内容,以完善系统的结构.在这个过程中,维护人员根据逆向恢复需求的变化,可以随时调整场景和迭代次数,增加流程的可控制性和适应性,以此提高维护活动结果的准确性.

同时,本文提出了利用关联规则挖掘来获取用况和事件的映射关系.通过概念格的分析,可以看出,关联规则可以挖掘出潜在的隐含关系,对建立映射关系有很大的帮助.通过对用况图的分析,可以直观的显示出用况在系统中的运行情况,结合具体类的模块调用情况,可以帮助调整和补充用况,更好的理解系统.

在实验中,通过分析包结构与实际运行系统的模块化结构之间的差异评估软件的结构质量.针对性能的差异我们进一步调查分析每个簇和实际系统的包,通过比较差异,为程序理解和可能的重构工作提供帮助.从实验的结果看,该方法还受限于很多因素的制约:如用况场景的构造,关联规则过滤阈值的设置,聚

类算法的切割值等主观因素,这些都需要通过实验数据和经验的积累来改善.

参考文献

[1] Chenchen Xiao, Vassilios Tzerpos. Software clustering based on dynamic dependencies [A]. 9th European Conference on Software Maintenance and Reengineering (CSMR)[C]. UK, Proceedings. IEEE Computer Society, 2005. 124 – 133.

[2] Chiragkumar Patel, Abdelwahab Hamou-Lhadj, et al. Software clustering using dynamic analysis and static dependencies [A]. 13th European Conference on Software Maintenance and Reengineering (CSMR)[C]. Kaiserslautern, Germany. IEEE Computer Society, 2005. 27 – 36.

[3] Nicolas Anquetil, Timothy Lethbridge. Experiments with clustering as a software remodularization method [A]. 6th Working Conference on Reverse Engineering (WCRE)[C]. Atlanta, Georgia, USA. IEEE CS, 1999. 235 – 255.

[4] Thomas Eisenbarth, Rainer Koschke, Daniel Simon. Locating features in source code [J]. IEEE Trans Software Eng, 2003, 29(3):210 – 224.

[5] Bernhard Ganter, Rudolf Wille. Formal Concept Analysis: Mathematical Foundations [M]. NJ, USA: Springer-Verlag New York, Inc Secaucus, 1999. 129 – 244.

[6] W Zhihua, V Tzerpos. An effectiveness measure for software clustering algorithms [A]. 12th International Workshop on Program Comprehension (IWPC)[C]. Bari, Italy: IEEE Computer Society, 2004. 194 – 203.

[7] JHotDraw Project [CP/OL]. http://www.jhotdraw.org, 2013-03-06.

[8] Dynamo (Dynamic Aspect Mining Tool) [CP/OL]. http://star.fbk.eu/dynamo/, 2013-03-06.

[9] Julien Repond, Philippe Dugerdil, Pietro Descombes. Use-case and scenario meta-modeling for automated processing in a reverse engineering tool [A]. 4th Annual India Software Engineering Conference (ISEC)[C]. Kerala, India: ACM Press, 2011. 135 – 144.

作者简介

任 武 男,高级工程师,博士.主要研究方向为软件工程、领域工程、信息安全等.

E-mail:081024014@fudan.edu.cn