基于文法产生式优化的设计模式识别方法

肖卓宇[1]，何 铠[2,3,4]，杨鑫维[1]，杨邦平[1]，陈俊旭[1]，胡振涛[1]

(1. 中南林业科技大学涉外学院 长沙 410200; 2. 广州大学计算机科学与教育软件学院 广州 510006;
3. 长沙理工大学计算机与通信工程学院 长沙 410114; 4. 北京大学高可信软件技术教育部重点实验室 北京 海淀区 100871)

【摘要】以精确的设计模式检测结果为目标，为解决设计模式识别的变体问题，提出一种基于文法产生式优化的设计模式识别方法，旨在使用可视化文法产生式描述设计模式参与者角色的属性与联系，并通过增加文法产生式描述的特征信息来识别重叠的设计模式及参与者角色间的附加关系。实验结果表明，该方法减少了设计模式识别的假阳性结果与假阴性结果，通过与主流方法的*F*-score评估指标比较，显示出该方法的优势。

关 键 词 设计模式检测; 文法产生式; 模式共享; 变体; 可视化
中图分类号 TP311 **文献标志码** A doi:10.3969/j.issn.1001-0548.2017.03.015

An Optimization Method for Design Pattern Identification Based on the Grammar Production

XIAO Zhuo-yu[1], HE Pei[2,3,4], YANG Xin-wei[1], YANG Bang-ping[1], CHEN Jun-xu[1], and HU Zhen-tao[1]

(1. Swan College, Central South University of Forestry and Technology Changsha 410200;
2. School of Computer Science & Education Software, Guangzhou University Guangzhou 510006;
3. School of Computer and Communication Engineering, Changsha University of Science and Technology Changsha 410114;
4. Key Laboratory of High Confidence Software Technologies of Ministry of Education, Peking University Haidian Beijing 100871)

Abstract Aiming at obtaining the accurate detection results of design pattern, an optimization method for design pattern identification based on the grammar production is presented for solving the variant problem of design pattern. The method focuses on role attributes of participants and relationships of design pattern instances by visual grammar production, and identifies the overlapping patterns and the additional relations between the participant roles by adding the feature information of grammar production. Experiment results show that the proposed method can reduce the false positive results and the false negative results. *F*-score index comparison with other well-known algorithms indicates the effectiveness and merits of the proposed method.

Key words design pattern detection; grammar production; pattern sharing; variant; visualization

设计模式恢复有助于软件的维护与程序的理解，常用于软件系统的重构[1]。当前众多研究在设计模式识别方面做出了重要贡献[2-3]。文献[4]提出将设计模式抽取的结果分类，并形成Clue、EDP等微结构。文献[5]通过分类学习来提升设计模式识别效果。文献[6]以子图同构的形式检测设计模式。文献[7]使用DSL方法以图形的形式检测设计模式。文献[8]通过注释对设计模式变体问题进行研究。文献[9]采用多阶段逐步筛选设计模式候选参与者集的方法来检测设计模式实例。文献[10]给出了多种影响设计模式检测精确率的原因，并提出了融合精确度与召回率的*F*-score指标。文献[11]对设计模式参与者角色间附加关系等导致的模式变体问题进行了初步研究。

总而言之，传统方法存在4类问题：

1) 设计模式识别在评估指标选择上没有同时兼顾假阳性与假阴性结果，以至于不能反应出真实的识别效果。

2) 评估结果仅单独分析比较召回率或精确率，缺乏一个综合的评估指标。

3) 对附加关系及参与者角色共享设计模式导致的变体问题缺乏深入研究，导致不精确结果产生。

4) 不能识别复杂多层类之间的设计模式实例。

为此，本文提出一种基于文法产生式优化的设

收稿日期：2015 – 12 – 28；修回日期：2016 – 05 – 26
基金项目：国家自然科学基金(61170199)；湖南省大学生研究性学习和创新性实验计划(湘教通[2015]84号197)；湖南省教学改革资助项目(湘教通[2016]400号1068)；广东省自然科学基金(2015A030313501)；广东省普通高校创新团队建设项目(2015KCXTD014)；湖南省教育厅重点基金项目(11A004)
作者简介：肖卓宇(1979 –)，男，副教授，主要从事程序理解、逆向工程、演化计算等方面的研究.

计模式识别方法，主要思路是将抽取后的信息通过文献[12]给出的可视化语言及文献[13]提出的文法以产生式的形式进行描述，并依据文献[14]中提出的47种特征信息进行表示。之后，增加文法产生式解决设计模式参与者角色共享设计模式实例及其附加关系导致的变体问题。最后，通过Jhotdraw等4个开源系统进行了实验设计与评估。

1 模式实例识别步骤

设计模式实例识别的5个步骤：

1) 抽取软件系统相关信息；

2) 以可视化文法的形式表示抽取结果；

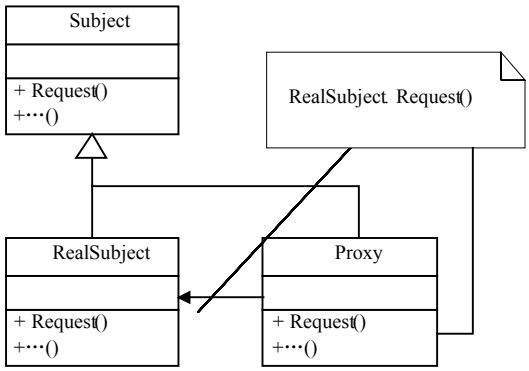
3) 将产生式表示为文献[14]给出的特征形式；

4) 增加可视化文法产生式识别设计模式变体；

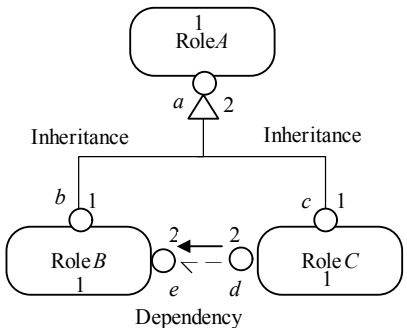
5) 通过工具Sparx Enterprise Architect(EA)[8]进行设计模式检测；

2 可视化文法表示设计模式实例

文献[12]通过可视化文法表示设计模式参与者角色间的属性及关系。如图1a中Proxy模式的Subject、RealSubject、Proxy类及关系实体Dependency、Inheritance可表示为可视化图形，而可视化图形的属性可通过连接点联系。



a. Proxy模式



b. Proxy可视化表示

图1 Proxy模式

图1b比图1a更加侧重参与者角色属性之间的联系，圆角矩形表示设计模式参与者角色，如Role*A*、Role*B*等，圆角矩形中的连接区域用数字进行编号，此外，圆角矩形周边的空心小圆圈表示角色属性的连接点，也用数字进行编号，参与者角色之间通过连接点衔接。如图1b中Role*C*与继承关系实体间通过角色Role*C*的属性区域1及连接点1在属性*c*处联系，而角色属性*d*表示Role*C*与Dependency关系之间通过角色Role*C*的属性区域1及连接点2进行联系。表1给出了图1b中参与者角色的属性表。

文献[13]依据文献[12]提出的可视化文法将图1b表示为字符串形式，其实质是通过定义约束来交替参与者角色及其属性，从而挖掘角色间的关系。

表1 参与者角色属性表

参与者角色	属性1	属性2	…	属性*n*
Role*A*	*a*			
Role*B*	*b*	*e*		
Role*C*	*c*	*d*		
Dependency	*d*	*e*		
Inheritance	*a*	*b*		
Inheritance	*c*	*d*		

定义 1 Role[*m*] $\text{Connect}_{i,j}^{k}$ Role[*n*]

定义1中*m*和*n*表示参与者角色的编号，$\text{Connect}_{i,j}$表示角色Role[*m*]的连接点或属性区域*i*与角色Role[*n*] 的连接点或属性区域*j*之间的联系，*k*表示参与者角色间不同关系的编号。

图1b中角色Role*C*的属性区域1与Inheritance关系实体的连接点1存在联系，可表示为：

$$\text{Role}C\ \text{Connect}_{1,1}\ \text{Inheritance} \tag{1}$$

此外，角色RoleB的属性区域1与Inheritance实体的连接点1也存在联系。依据定义1，为了避免与式(1)的$\text{Connect}_{1,1}$符号冲突，式(2)中的*k*值取2，有：

$$\text{Role}B\ \text{Connect}_{1,1}^{2}\ \text{Inheritance} \tag{2}$$

定义 2 Role[*m*] $\{\text{Connect}_{i,j}^{k}\}$ Role[*m*]

定义2表示角色Role[*m*]与角色Role[*n*]的连接点或角色属性区域间各种联系组成的关系实体集。

依据定义1，图1b可表示为：

$$\text{Role}A\ \text{Connect}_{1,2}\ \text{Inheritance}\ \text{Connect}_{1,1}\ \text{Role}B\ \text{Connect}_{1,2}^{2}\ \text{Dependency}\ \text{Connect}_{2,1}\ \text{Role}C\ \text{Connect}_{1,1}^{2}\ \text{Inheritance}\ \text{Connect}_{2,1}^{2}\text{Role}A \tag{3}$$

图1b中圆角矩形表示模式的参与者角色，如Role*A*、Role*B*、Role*C*，每个圆角矩形中有相应的数字表示角色的属性区域，而其周边的小圆圈表示参与角色属性区域与关系实体间的连接点，用数字标

注连接点区分。式(3)中Role*A*角色的属性区域1与Inheritance实体的圆形连接点2在表1中的角色属性*a*处连接，可表示为$\mathrm{Connect}_{1,2}$。此外，为了区分已经存在的$\mathrm{Connect}_{1,2}$，Role*B*角色的属性区域1与Dependency关系实体的连接点2在表1的角色属性*b*处相连，可用$\mathrm{Connect}_{1,2}^{2}$表示。

依据定义1与定义2，在文献[13]提出的文法基础上进行改进，旨在实现文法产生式终结符与非终结符角色的替换。为此，Proxy模式可表示为：

$$\begin{aligned}&\mathrm{Proxy\_pattern}\rightarrow \mathrm{Subject}\ \ \mathrm{Connect}_{1,2}\ \ \mathrm{Inheritance}\\&\mathrm{Connect}_{1,1}\mathrm{RealSubject}\ \mathrm{Connect}_{1,2}^{2}\mathrm{Delegation}\ \mathrm{Connect}_{2,1}\\&\mathrm{Proxy}\ \mathrm{Connect}_{1,1}^{2}\ \ \mathrm{Inheritance}\ \ \mathrm{Connect}_{2,1}^{2}\ \ \mathrm{Subject}\end{aligned}\tag{4}$$

$$\mathrm{Subject}\rightarrow \mathrm{Role}\ \Delta:\{\mathrm{Subject}_i::=\mathrm{Role}_i\}\tag{5}$$

$$\mathrm{RealSubject}\rightarrow \mathrm{Role}\ \Delta:\{\mathrm{RealSubject}_i::=\mathrm{Role}_i\}\tag{6}$$

$$\mathrm{Proxy}\rightarrow \mathrm{Role}\ \Delta:\{\mathrm{Proxy}_i::=\mathrm{Role}_i\}\tag{7}$$

式(4)描述了Proxy模式，式(5)～式(7)使用Δ规则[12]依次将参与者扮演的角色通过产生式左右两侧的非终结符与终结符替换出来。如式(7)中Role角色扮演了Proxy模式中的Proxy角色，Role角色的属性*i*也对应了Proxy角色的属性*i*。

图2中Proxy模式的Subject、RealSubject、Proxy角色通过Δ规则及式(5)～式(7)依次被扮演的角色替换出来，详见步骤2)～步骤4)，而矩形点线框表示替换后的结果，最终，经历步骤5)之后，一个标准的Proxy模式被识别出来。

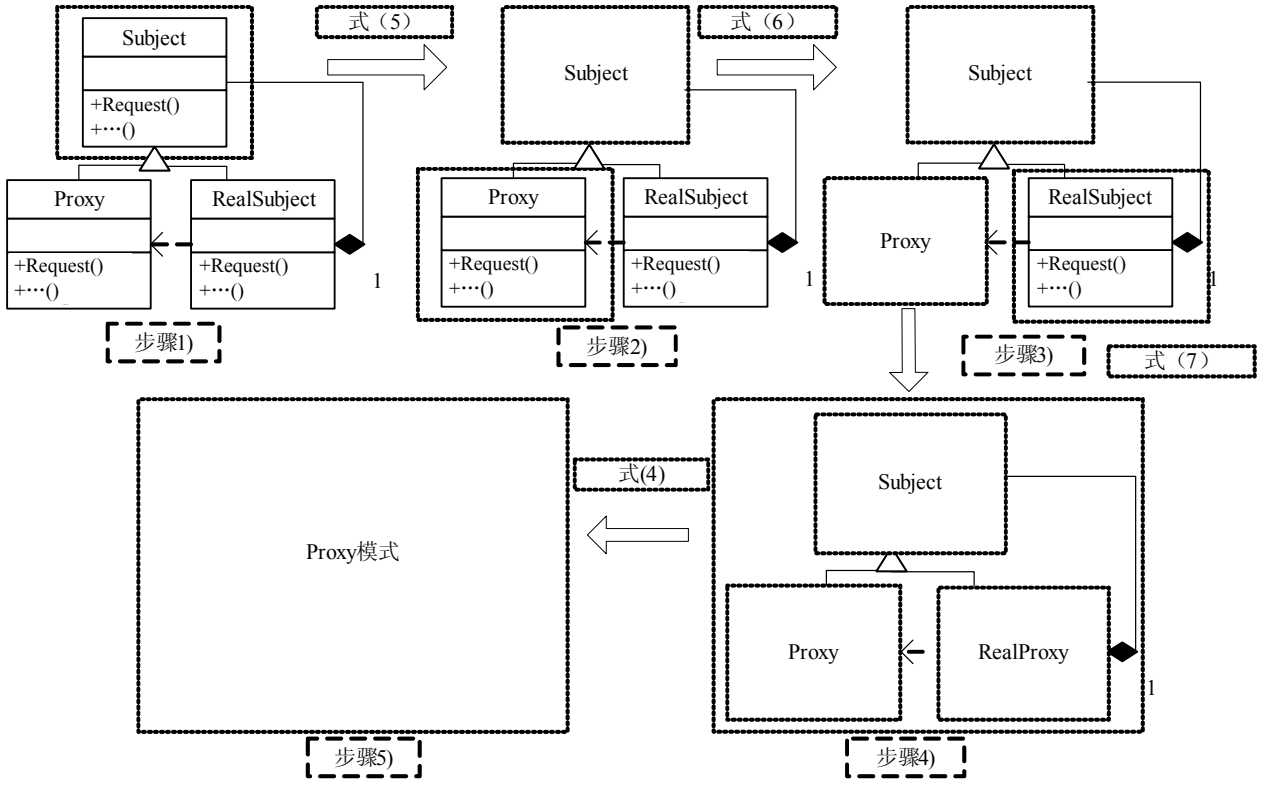


图2　Proxy模式检测流程图

3　设计模式实例变体检测

式(4)～式(7)在一定程度上解决了参与者在设计模式中的角色扮演问题，但没有考虑设计模式参与角色共享及设计模式角色间附加关系导致的设计模式变体问题，这将导致模式检测时产生大量假阳性与假阴性结果[11]。传统的方法较少考虑变体问题，即使一部分方法关注了变体，但仅单方面考虑变体结果的假阴性问题或假阳性问题，缺乏将二者结合进行深入的研究与分析，导致了不精确的检测结果出现。

为解决这个问题，本文研究兼顾设计模式变体的假阳性与假阴性结果检测，对文献[13]提出的文法进行改进，并表示为文献[14]中给定的47种特征，通过工具Sparx Enterprise Architect(EA)[8]实现设计模式实例识别。表2描述了文献[14]中部分重要特征。

表2 特征类型表

特征名称	特征描述
GetAllClass(*C*1,*C*2,⋯,*Cn*)	获得所有的抽取类
GetAllInterfaces(*C*1,*C*2,⋯,*Cn*)	获得所有的抽取接口
Has Generalization(*C*1,*C*2)	*C*1与*C*2角色存在泛化关系
Has Delegation(*C*1,*M*1,*C*2,*M*2)	*C*1的*M*1方法调用了*C*2的*M*2
NotHas Association(*C*1,*C*2)	*C*1与*C*2角色不存在关联关系
Has Annotation(*I*,*C*,*M*,*S*)	接口、类、方法、申明有注释
HasCommon-Operation(*C*1,*C*2,*M*)	*C*1,*C*2角色有共同的方法*M*
NotHas Aggregation(*C*1,*C*2)	*C*1与*C*2角色不存在聚合关系
Has Parameter(*M*,*Pi*)	方法*M*有*P*1,*P*2,⋯,*Pn*个参数
Has Return Value(*M*,*V*)	方法*M*有一个返回值*V*
Has Static Instance(*A*)	属性*A*是静态的
HasCommonChildClasses(*C*1,*Cn*)	*C*1是父类，有*n*个相同的子类
Has SameSignature(*M*1,*M*2)	方法*M*1与*M*2有相同的签名
⋮	

3.1 设计模式假阴性结果检测

传统方法检测设计模式时，图3不能识别为Proxy模式，究其原因发现，传统方法要求Proxy模式中AbstractRealSubject类的方法Request需能被Proxy类的Request方法委托实现，而AbstractRealSubject在图3a中是一个抽象类，其方法Request需调用其子类RealSubject1或RealSubject2的方法Request实现。事实上这是一个典型的Proxy模式变体，因为Proxy类中的方法Request最终还是能够通过调用AbstractRealSubject子类中的方法实现的，属于Proxy模式的变体，这是一个典型的假阴性结果。

除此之外，图3a点线框中的3个类RealSubject1、RealSubject2、AbstractRealSubject及其关系也是一个Composite模式，但传统的方法也不能识别，也属于一个典型的假阴性结果。对图3a进一步研究发现，AbstractRealSubject这个类参与者角色被Proxy与Composite两种设计模式共享，即扮演了Proxy模式中的RealSubject角色，又扮演了Composite模式中的Component角色。

传统检测方法很难识别涉及复杂多层重叠的设计模式。为检测图3a中的Proxy变体及Composite与Proxy与的重叠问题，通过在第2节提出的文法基础上增加产生式实现。

$$\text{DesignPatterns} \rightarrow \text{ShareRole} \mid \text{Design\_Pattern} \quad (8)$$

$$\text{Design\_Pattern} \rightarrow \text{Proxy\_pattern}|\text{Composite\_pattern}|... \quad (9)$$

$$\text{ShareRole} \rightarrow \text{bstractRealSubject Connect}_{1,2}\text{Delegation Connect}_{2,1}\text{ProxyConnect}^{2}_{1,1}\text{InheritanceConnect}^{2}_{2,1}\text{Subject Connect}^{2}_{1,2}\ \text{Inheritance}\ \text{Connect}_{1,1}\ \text{ShareRole} \quad (10)$$

$$\text{Subject} \rightarrow \text{Role}\ \ \Delta:\{\text{Subject}_i::=\text{Role}_i\} \quad (11)$$

$$\text{Proxy} \rightarrow \text{Role}\ \Delta:\{\text{Proxy}_i::=\text{Role}_i\} \quad (12)$$

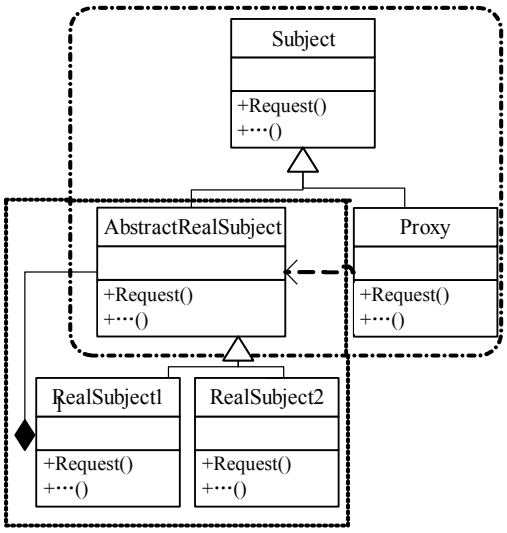
$$\text{RealSubject} \rightarrow \text{ShareRole}\ \ \Delta:\{\text{RealSubject}_i::=\text{ShareRole}_i\} \quad (13)$$

$$\text{ShareRole} \rightarrow \text{AbstractRealSubject Connect}_{1,3}\text{Inheritance Connect}_{1,1}\text{RealSubject2Connect}^{2}_{1,1}\text{RealSubject1Connect}_{1,2}\ \text{Aggregation Connect}_{4,1}\ \text{ShareRole} \quad (14)$$

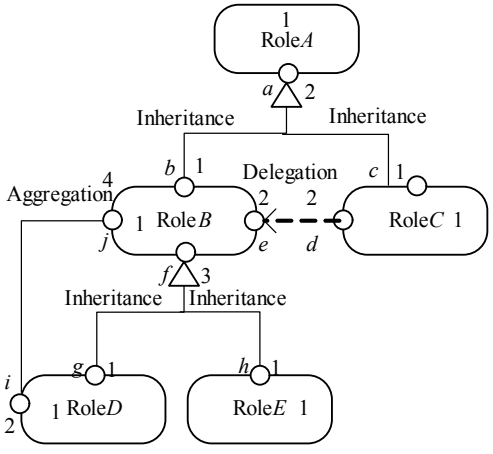
$$\text{AbstractRealSubject} \rightarrow \text{Role}\ \ \Delta:\{\text{AbstractRealSubject}_i::=\text{Role}_i\} \quad (15)$$

$$\text{RealSubject1} \rightarrow \text{Role}\ \Delta:\{\text{RealSubject1}_i::=\text{Role}_i\} \quad (16)$$

$$\text{RealSubject2} \rightarrow \text{Role}\ \Delta:\{\text{RealSubject2}::=\text{Role}_i\} \quad (17)$$



a. Proxy模式变体



b. Proxy模式变体可视化表示

图3 Proxy模式变体

式(8)中ShareRole表示参与多个设计模式实例的共享角色，Design_Pattern表示被识别出的设计模式；式(9)中Design_Pattern这个非终结符可被23种标准的GOF设计模式终结符替换。式(11)～式(13)通过Δ规则依次实现对Proxy模式的角色替换。

表3　设计模式实例角色扮演对应表

Proxy 模式		Composite 模式	
Proxy:	Proxy	Component:	AbstractRealSubject
Subject:	Subject	Composite:	RealSubject1
RealSubject:	AbstractRealSubject	Leaf:	RealSubject2

表3描述了具体的映射关系，Proxy、Subject、AbstractRealSubject依次扮演了Proxy模式中的Proxy、Subject、Real Subject角色，其中加边框与灰色底纹的AbstractRealSubject类除扮演了Proxy模式的RealSubject角色外，还扮演了Composite模式中的Component角色，而字体加粗部分表示被扮演的设计模式角色。

首先通过产生式识别出图4中圆角矩形虚线框中的Proxy模式，见步骤2)。此外，式(14)中共享角色ShareRole再次作为非终结符起点对可能存在关联的角色及关系继续进行遍历，最终，式(15)～式(17)通过Δ规则依次实现对Composite模式角色替换，表2中AbstractRealSubject、RealSubject1、RealSubject2依次扮演了Composite模式的Component，Composite，Leaf角色。最终图4中点线矩形框中的Composite被识别出来，见步骤3)。

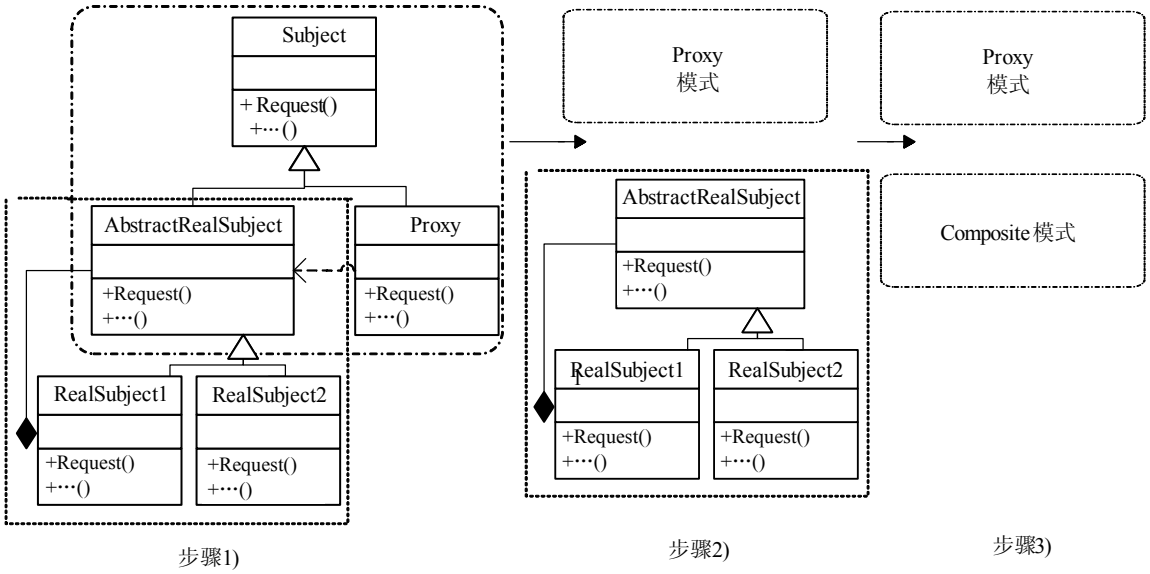


图4　重叠设计模式识别流程图

表4　Proxy特征表示

行	Proxy特征表示
1	GetAllClass(Proxy, AbstractRealSubject, RealSubject1, RealSubject2)
2	GetAllInterfaces(Subject)
3	Has Generalization(Proxy,Subject)
4	Has Generalization(AbstractRealSubject, RealSubject)
5	NotHas Association(Proxy, AbstractRealSubject)
6	Has Delegation(Proxy, request, RealSubject, request)
7	Has Common-Operation(Proxy, Subject, RealSubject, request)

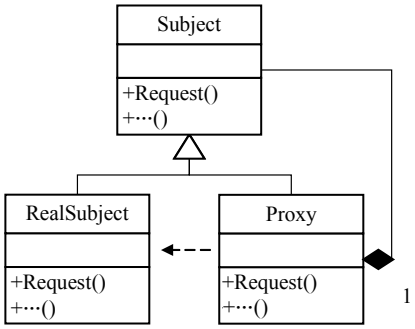
依据表2给出的特征，表4描述了Proxy模式文法产生式(8)～式(12)的具体实现。第1行GetAllClass表示Proxy模式中存在Proxy, AbstractRealSubject, Real-Subject1,RealSubject2共4个类角色，第2行表示Subject角色是接口，第3、4行描述了Proxy与Subject，AbstractRealSubject与RealSubject两组角色间存在泛化关系，第5行限定Proxy与AbstractRealSubject不能存在关联关系，第6行描述了Proxy的request方法与RealSubject的request方法通过委托实现，第7行表示角色Proxy、Subject及RealSubject中有同名方法request。

3.2　设计模式假阳性结果检测

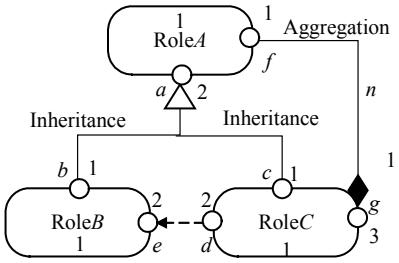
传统方法检测设计模式时，图5易被错误的识别为Proxy模式，这是一个典型的假阳性结果，因为标准Proxy模式中的Proxy类和Subject不存在聚合关系，这是一类典型的附加关系[11]。

规则1：Proxy类不能与Subject类存在聚合关系。

为检测图5中的Proxy模式的假阳性问题，仍需对第2节文法进行优化，旨在通过增加产生式来检测规则1。为此增加一个附加关系检测标记access，若设计模式识别过程中发现了规则1描述的附加关系，则access值取1，以便后续阶段检测过程中忽略该关系的搜寻，从而避免假阳性结果。Tagging表示附加关系是否关联其他参与者角色的属性，初值设为0。



a. Proxy模式变体



b. Proxy模式变体可视化表示

图5 Proxy模式变体

此外，引入文献[12]提出的Γ规则对产生式进行约束，式(20)中Role′角色的access标志为1，表示Proxy角色存在附加关系Aggregation，故后续遍历将不再检测该关系。相反，若access值设为0，则式(20)中的Tagging不用标记，这表示违反了规则1的约束。

$$\text{Proxy\_pattern} \rightarrow \text{Role}A\ \text{Connect}_{1,2}\text{Inheritance}\ \text{Connect}_{1,1}\ \text{Role}B\ \text{Connect}^{2}_{1,2}\ \text{Dependency}\ \text{Connect}_{2,1}\ \text{Role}C\ \text{Connect}^{2}_{1,1}\ \text{Inheritance}\ \text{Connect}^{2}_{2,1}\ \text{Role}A \tag{18}$$

$$\text{Subject} \rightarrow \text{Role}\ \Delta:\{\text{Subject}_i ::= \text{Role}_i\} \tag{19}$$

$$\text{Proxy} \rightarrow \text{Proxy}'\ \{\text{Connect}_{1,3}, \neg\text{Tagging}\}\ \text{Aggregation}\ \text{Connect}_{1,1}\ \text{Role}A$$

$$\Delta:\{\text{Proxy}_1 ::= \text{Proxy}'_1\}$$

$$\Gamma:\{(\text{Role}':\text{Role}'_1 ::= \text{Role}_1; \text{Role}'_{\text{access}} ::= 1)\} \tag{20}$$

$$\text{RealSubject} \rightarrow \text{Role}\ \Delta:\{\text{RealSubject}_i ::= \text{Role}_i\} \tag{21}$$

$$\text{Proxy} \rightarrow \text{Role}\ \Delta:\{\text{Proxy}_i ::= \text{Role}_i\} \tag{22}$$

文法产生式(18)～式(22)依据表2的特征描述表示为表5，其中表5的第1行～第7行描述了一个标准的Proxy模式，而第8行是与标准的Proxy模式相比发生了变化的特征，即Proxy与Subject类之间存在附加的聚合关系。此外，第1行GetAllClass表示Proxy模式中存在Proxy、Abstract- RealSubject等类，第2行表示Subject角色是接口，第3,4行描述了Proxy与Subject、RealSubject与Subject两组角色间存在泛化关系，第5行限定Proxy与RealSubject不能存在关联关系，第6行描述了Proxy的request方法与RealSubject的request方法通过委托实现，第7行表示角色Proxy、Subject、RealSubject中有同名方法request。

表5 Proxy假阳性变体特征表示

行	Proxy假阳性变体特征表示
1	GetAllClass(Proxy, AbstractRealSubject, RealSubject)
2	GetAllInterfaces(Subject)
3	Has Generalization(Proxy, Subject)
4	Has Generalization(RealSubject, Subject)
5	NotHas Association(Proxy, RealSubject)
6	Has Delegation(Proxy, request, RealSubject, request)
7	Has Common-Operation(Proxy, Subject, RealSubject, request)
8	Has Aggregation(Proxy, Subject)

4 实验与结果分析

为评估基于文法产生式优化的设计模式识别方法有效性，选用JavaAWT等4个开源系统展开实验。评估通过真阳性、假阳性、假阴性结果来关注精确率与召回率，并通过*F*-score指标进行了比较试验。

1) 真阳性TP: 正确的模式识别为正确；

2) 假阳性FP: 错误的模式识别为正确；

3) 假阴性FN: 正确的模式识别为错误；

执行实验的操作系统采用WINDOW8，CPU为INTEL Core4 E3-1230 V2 3.3 GHz，内存16 GB。

$$P = \frac{\text{TP}}{\text{TP}+\text{FP}} \tag{23}$$

$$R = \frac{\text{TP}}{\text{TP}+\text{FN}} \tag{24}$$

式中，*P*为精确率；*R*为召回率。

表6中第一列字段表示设计模式类型，依据GOF原则分为了结构型(structure,S)，行为型(behavior, B)，创建型(creation,C)3类[1]，此外，(not analysed , NA)表示该模式在系统中未发现。通过分析表6中4个系统的TP、FP、FN值，发现了几个明显的问题。

1) 结构型模式S相对行为型模式B与创建型模式C更容易产生假阳性与假阴性结果。深入研究后发现，一方面行为型模式与创建型模式在4个开源系统中使用的数量与频率相对结构型模式低。另一方面由于行为型模式在执行时对复杂的控制流几乎无法跟踪，而创建型模式侧重抽象实例化，并涉及了委托等复杂的关系，故行为型与创建型模式产生变体的几率相对结构型模式要小。

2) 4个开源系统的检测结果表明，Singleton是创建型模式中较易产生变体的模式，在javaAWT 5.0中甚至同时出现了假阳性结果FP与假阴性结果FN。究其原因发现Singleton模式仅保证系统中的一个类只有一个实例，这样的简单结构易于扩展，故容易产

生变体。为此，文献[11]分析了Singleton模式的8种变体。后续工作中将通过增加文法产生式并结合特征结构进一步改进现有工作。

3) composite、adapter、state等模式相对容易识别出假阳性与假阴性结果，在深入分析后发现，composite与decorator模式、state与stategy模式、adapter与command模式容易发生重叠。表7描述了JavaAWT 5.0与Dom4j 1.6.1系统识别后的模式重叠结果。为此，后续工作中将考虑将composite与decorator模式、state与stategy模式、adapter与command模式组合成3组进行检测试验，并通过复杂大类图分割法等优化方法避免消极的模式共享。

表6　4类开源系统设计模式识别表

类型	模式名	JhotDraw 6.0b1					Dom4j 1.6.1					JavaAWT 5.0					Apache ant v1.6.2				
		TP	FP	FN	*P*/%	*R*/%	TP	FP	FN	*P*/%	*R*/%	TP	FP	FN	*P*/%	*R*/%	TP	FP	FN	*P*/%	*R*/%
S	Proxy	17	1	1	94	94	21	1	2	95	91	27	1	10	96	94.2	9	0	1	100	90
S	Composite	27	1	0	96	100	55	2	0	96	100	108	3	2	97	98.1	11	0	0	100	100
S	Decorator	26	0	0	100	100	10	1	0	91	100	69	1	0	99	100	14	1	0	93	100
S	Adapter	33	1	3	97	92	62	3	2	95	97	394	3	15	99	96.3	145	0	4	100	97
B	Template	125	0	2	100	98	41	0	0	100	100	290	0	6	100	98.0	6	0	0	100	100
B	Visitor	4	0	1	100	80	12	0	1	100	92	13	0	2	100	86.7	NA	NA	NA	NA	NA
B	state	4	0	1	100	80	29	1	2	97	94	134	2	12	99	91.8	NA	NA	NA	NA	NA
C	Factory	44	1	0	98	100	53	0	1	100	98	55	0	2	100	96.5	38	0	0	100	100
C	Prototype	3	0	0	100	100	16	0	0	100	100	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
C	Singleton	6	0	1	100	86	32	0	4	100	89	135	1	11	99	92.5	7	0	2	100	80
	平均统计				98.5	93				97.4	96.1				98.8	94.9				98	95.3

表7　JavaAWT 5.0与Dom4j 1.6.1系统重叠模式检测表

类型	模式名	JavaAWT 5.0			Dom4j 1.6.1		
		TP	FP	FN	TP	FP	FN
S	Proxy	1	0	1	1	0	1
S	Composite	4	1	1	4	2	0
S	Decorator	2	1	0	1	1	0
S	Adapter	8	0	1	3	1	2
B	Template	3	0	0	2	0	0
B	Visitor	0	0	2	0	0	1
B	state	5	2	2	3	1	1
C	Factory	1	0	2	1	0	1
C	Prototype	NA	NA	NA	0	0	0
C	Singleton	3	1	1	2	0	1

如：Peterson从召回率与精确率两个角度出发提出一种综合的评估指标*F*-score：

$$F\text{-score}=\frac{(1+W^2)PR}{W^2P+R} \tag{25}$$

式中，权值W取2.8。*F*-score值越大则表明设计模式检测效果越好，为此，将本文方法与文献[2,6-9]的识别方法通过Jhotdraw进行了*F*-score指标比较实验，表8结果说明了本文方法值得进一步深入研究。

表8　Jhotdraw开源系统中的*F*-score指标值对比

检测方法	平均精确度/%	平均召回率/%	*F*-score /%
本文方法	98.5	93	94
文献[2]	93	63	65
文献[6]	92	69	71
文献[7]	71	84	82
文献[8]	92	74	76
文献[9]	42	100	86

5　结 束 语

本文对设计模式实例识别的精确性问题进行了研究，提出一种基于文法产生式优化的设计模式识别方法，该方法通过逆向工程工具抽取特征信息，在文献[12]提出的可视化语言基础上结合文献[13]的文法，以文献[14]给出的特征信息对设计模式实例进行了描述，并通过增加文法产生式优化了识别结果。实践结果表明新方法具有较好精确率，解决了模式实例检测结果的假阳性及假阴性问题。

参 考 文 献

[1] KACZOR O, GUEHENEUC Y G, HAMEL S. Identification of design motifs with pattern matching algorithms[J]. Information and Software Technology, 2010, 52(2): 152-168.

[2] AMPATZOGLOU A, CHARALAMPIDOU S, STAMELOS I. Research state of the art on GoF design patterns: a mapping study[J]. Journal of Systems and Software, 2013, 86(7): 1945-1964.
[3] YU D, ZHANG Y, CHEN Z. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures[J]. Journal of Systems and Software, 2015, 88(5): 1-16.
[4] FONTANA F A, MAGGIONI S, RAIBULET C. Design patterns: a survey on their micro-structures[J]. Journal of Software: Evolution and Process, 2013, 25(1): 27-52.
[5] ZANONI M, FONTANA F A, STELLA F. On applying machine learning techniques for design pattern detection[J]. Journal of Systems and Software, 2015, 88(5): 102-117.
[6] DONG J, ZHAO J, SUN Y. A matrix based approach to recovering design patterns[J]. IEEE Transactions on Systems, Man and Cybernatics, 2009, 39(6): 1271-1282.
[7] BERNARDI M L, CIMITILE M, DI LUCCA G.. Design pattern detection using a DSL-driven graph matching approach[J]. Journal of Software: Evolution and Process, 2014, 26(12): 1233-1266.
[8] RASOOL G, PHILIPPOW I. Design pattern recovery based on annotations[J]. International Journal of Advances in Engineering Software, 2010, 41(4): 519-526.
[9] GUEHENEUC Y G, ANTONIOL G. DeMIMA: a multilayered approach for design pattern identification[J]. IEEE Transactions on Software Engineering, 2008, 34(5): 667-684.
[10] PETTERSON N, LOWE W, NIVRE J. Evaluation of accuracy in design pattern occurrence detection[J]. IEEE Transactions on Software Engineering, 2010, 36(4): 575-590.
[11] STENCEL K, WEGRZYNOWICZ P. Implementation variants of the singleton design pattern[C]//On the Move to Meaningful Internet Systems: OTM 2008 Workshops. Berlin Heidelberg: Springer, 2008: 396-406.
[12] COSTAGLIOLA G, DE LUCIA A, OREFICE S, et al. A classification framework to support the design of visual languages[J]. Journal of Visual Languages and Computing, 2002, 13(6): 573-600.
[13] COSTAGLIOLA G, DE LUCIA A, OREFICE S, et al. A parsing methodology for the implementation of visual systems[J]. IEEE Transactions on Software Engineering, 1997, 23(12): 777-799.
[14] RASOOL G, MADER P. A customizable approach to design patterns recognition based on feature types[J]. Arabian Journal for Science and Engineering, 2014, 39(12): 8851-8873.

编 辑 税 红