源代码中设计模式实例的抽取及验证方法研究*

李文锦[a], 王康健[b]
(中国计量学院 a. 现代科技学院; b. 信息工程学院, 杭州 310018)

摘 要: 从源码中抽取设计模式对于提高软件可理解性和可维护性、软件设计重用以及软件重构具有重要意义。面向 Java 语言提出了一种静态和动态分析相结合的源码中设计模式的抽取方法。具体地,研究了源码中设计模式抽取的静态结构分析过程,为了进一步提高设计模式实例抽取的准确率,对结构分析得到的创建型模式候选,使用创建对象的多重性分析方法进行验证,对结构分析得到的行为型模式候选,使用动态分析的方法进行验证,以区分结构相似但行为不同的模式的实例。最后实现了设计模式抽取工具并对开源软件中的模式实例进行抽取。通过实验数据,验证了设计模式实例抽取及验证方法的可行性及有效性。

关键词: 设计模式; 逆向工程; 多重性分析; 动态分析
中图分类号: TP311 **文献标志码**: A **文章编号**: 1001-3695(2012)11-4199-07
doi:10.3969/j.issn.1001-3695.2012.11.050

Research on detecting and validating design pattern instances from source code

LI Wen-jin[a], WANG Kang-jian[b]
(*a. College of Modern Science & Technology, b. College of Information Engineering, China Jiliang University, Hangzhou* 310018, *China*)

Abstract: Identifying design patterns from source code is one of the most promising methods for improving software maintainability, reusing experience and facilitating software refactoring. This paper presented an approach that combined static and dynamic analysis on detecting design patterns from Java source code and illustrated the static structural analysis phase of identifying pattern candidates. Especially, to improve the detecting precision, the approach executed multiplicity analysis of object creation to validate creational pattern candidates and dynamic analysis to validate behavior pattern candidates. After the validation of behavior candidates, the approach distinguished those instances belonged to patterns which had similar structures but different behaviors successfully. Finally, this paper implemented a tool of extracting design pattern from Java source code. The experiment results of applying it on an open source software show the feasibility and availability of the approach.

Key words: design pattern; reverse engineering; multiplicity analysis; dynamic analysis

0 引言

设计模式是面向对象设计的一个高级抽象,从程序理解和软件维护的观点出发,一个设计模式提供了模式结构中每个类的角色信息和模式各组成元素的关系以及模式组成元素和系统其余部分关系的信息。因此,源码中设计模式的抽取是逆向工程中的一个关键问题。对于缺少分析和设计文档的软件系统来说,从它的源码中抽取设计模式有助于软件系统的理解和文档化,增强软件系统的可维护性。进一步,可以将此方法应用于识别软件系统中引入设计模式后可以改进的地方并辅助进行相应的修改,提高源码质量,从而达到软件重构的目的。

由于 Java 语言在各种软件开发领域中的广泛使用,本文面向 Java 语言提出一种设计模式的抽取方法,首先提出结合了静态分析和动态分析方法的设计模式抽取的框架,对使用静态分析方法得到源码中的模式候选实例的过程进行简要说明,重点提出了在结构化分析得到模式实例之后,使用数据流分析验证创建型模式实例及使用动态分析验证行为型模式的方法。

本文介绍了本领域研究的相关工作和从 Java 软件抽取设计模式的方法框架。详细讨论了使用数据流分析的方法验证创建型模式实例的过程和使用动态分析的方法验证行为型模式实例的过程,并给出模式抽取工具应用于 ASM[1] 软件得到的实验结果。

1 相关工作

目前,从源码中抽取设计模式技术的研究在国际上受到了普遍关注。单纯从分析方法的角度划分大致分为三类:

a)只分析静态结构信息的方法。Tsantalis 等人[2] 使用相似度评分方法查找设计模式实例是静态分析方法的代表。在该方法中,首先将目标系统和设计模式中的信息,如类之间的关联关系、继承关系、抽象类、对象的创建及相同的抽象方法的调用等均表示成矩阵形式;其次将目标系统划分为以继承层次为核心的子系统,然后使用相似度算法计算子系统和模式相应的各种矩阵的相似程度;最后设计模式的实例由与模式角色相似度最高的子系统的类组合而成。这种方法的特点是支持目

收稿日期: 2012-04-08; **修回日期**: 2012-05-22 **基金项目**: 国家自然科学基金资助项目(61100160);浙江省教育厅科研项目(Y201018837)

作者简介:李文锦(1980-),女,吉林白山人,讲师,硕士研究生,主要研究方向为逆向工程(liwenjin@cjlu.edu.cn);王康健(1979-),男,浙江杭州人,副教授,博士研究生,主要研究方向为计算机图形学、软件工程等.

标系统中设计模式变体的发现。由于仅仅使用静态分析的方法查找模式实例,对于行为型的设计模式只能提供一个候选的实例集合,需要进一步结合动态分析验证实例是否准确。

b)加入静态行为分析的方法。Shi 等人[3]对 GOF[4]中的23个设计模式从逆向抽取的角度进行了重新划分,分为程序设计语言提供的模式、结构驱动型模式、行为驱动型模式、领域特定型模式、一般概念型模式五类。该作者重点讨论了结构驱动模式和行为驱动模式的查找方法。对于结构驱动型模式,主要通过分析目标系统的静态结构信息来查找模式实例。对于行为驱动型模式,首先分析目标系统的静态结构信息得到候选实例;然后对每个候选实例的方法内部进行静态的行为分析(即数据流分析)以验证是否满足模式的行为特征。尽管使用静态行为分析的方法可以获得目标系统在运行期间可能发生的情况,但并不能描述在运行期间实际发生的方法调用及对象引用的具体类型等。

c)加入动态行为分析的方法。Wendehals 等人[5,6]倾向于模式行为信息的匹配。首先从设计模式的描述出发,使用UML 2.0 中的序列图来构建模式的行为模型,并将模式的序列图转换为确定有限自动机 DFA。模式实例的匹配是将一个源代码示例的执行路径与定义好的模式行为模型 DFA 进行比对的过程。查找的结果为 DFA 在执行时到达接收状态的次数除以 DFA 在执行时到达拒绝状态的次数,由软件工程师根据该结果进行判断是否作为模式的实例。Von Detten 等人[7]对上述方法进行了扩展,主要扩展了模式行为模型的描述,对于参与者是包含零至多个对象的集合的模式(如 observer 模式、chain of responsibility 模式等)引入了对象集合的概念和 each 片段进行描述。这种描述方法使得进行模式匹配时在集合对象的匹配上更加灵活。

Pettersson[8]对执行路径的覆盖率对于静态和动态模式识别的准确度的影响进行了讨论。该方法使用静态和动态方法相结合查找软件系统中的设计模式实例。在静态分析过程中,使用 Recoder 工具进行源码静态信息的抽取,把每一个设计模式描述成一个元组,元组由元组元素和这些元素之间的关系共同组成。利用 Crocopat 执行查询来查找源码中满足模式元组中元素关系约束的实例。在动态分析过程中,使用 JVMDI 执行源代码的动态分析,收集每个模式候选实例中方法调用的信息并将这些信息保存在日志文件中。对静态分析得到的每个模式候选实例创建一个动态分析器,来识别候选实例在执行过程中是否满足模式元素之间的动态约束条件。其中关于动态分析器的创建说明较少,而这又是动态分析模式行为较重要的一个环节。

Ng 等人[9]倾向于创建型和行为型模式的发现。该作者定义了一个场景图的元模型,并用该模型去描述设计模式的行为信息和软件系统的动态信息。首先,将创建型和行为型模式描述中包含的协作关系图转换成场景图元模型的一个实例。源代码的动态分析过程为:在一些场景下执行系统,通过使用工具 BCEL 监控方法的执行过程获得源代码的执行序列,从而得到源代码的场景图。创建型模式和行为型设计模式实例的查找转换为约束满足问题(CSP)。CSP 中的变量是设计模式模型中的类或对象以及消息,需要满足的约束是设计模式模型中各实体之间的联系。变量的值域为目标系统场景图中的实体。CSP 的求解过程由基于解释的约束求解工具 JCHOCO 完成。这种方法忽视了静态分析对设计模式实例查找的重要性。

DEMIMA[10]工具基于一个多层次的方法来识别源代码中的设计模式。第一层次构建源代码的模型,包括所有 Java 系统中的所有元素,如类、接口、方法等。第二层次识别类的特征及类之间的关系。在该层次,DEMIMA 给出了设计模式实现中包含的类之间关系(如关联、聚合、继承等)形式化定义。这些关系的定义中包含了四个特征:类实例的排他性、消息接收者的类型、类实例的生命周期及多重性。该阶段使用了路径分析技术获取类实例的排他性及生命周期等信息。测试用例的选择是否合理将影响动态分析的结果。第三阶段使用基于解释的约束编程和约束放松方法从源代码模型中识别设计模式实例。

Lucia 等人[11]在其静态分析工作的基础上[12],使用可视化语言解析的方法完成行为型模式的识别。设计模式动态信息使用含有语义动作的监控语法规则进行描述,这种监控语法规则从模式的 UML 序列图构建而来,并用于产生该模式的解析器。目标软件的动态分析过程为:使用 Probekit 工具对静态分析得到的模式候选实例相关的代码进行插桩,在合适的测试用例下执行插桩后的代码,得到需要监控的方法之间调用的序列图。设计模式实例的验证过程即为目标软件的方法执行序列作为解析器的输入并依次匹配消减监控语法规则中的产生式的过程。

与上述方法比较,本文的工作主要体现在以下几点:

a)提出结合了静态分析和动态分析方法的设计模式抽取过程的完整框架。

b)针对结构化分析得到的创建型模式候选实例,使用数据流分析的方法对实例创建的数量进行度量,以进一步验证创建型模式实例,提高这类模式识别的精度。

c)针对结构化分析得到的行为型模式候选实例,使用动态分析的方法进一步验证。动态分析过程中定义状态图模型并使用状态图统一进行模式行为的描述,提出模式候选实例的执行序列与模式状态图的匹配算法。

2 设计模式抽取框架

从源代码中抽取设计模式实例首先需要对 Java 编译结果(即 Java Bytecode)进行静态分析,得到结构型模式实例集合、创建型模式实例候选集合以及行为型实例模式的候选集合;其次对于抽取得到的创建型模式实例的候选,进行实例创建验证,验证通过的候选即为创建型模式实例;最后对于抽取得到的行为型模式实例的候选进行动态分析验证,验证通过的候选即为行为型模式实例,抽取过程如图1所示。

下面对每个步骤进行概要描述。

1) 静态分析 它首先将每个待抽取模式进行形式化,然后分析源代码内容并获取源代码内部表示,最后将二者进行匹配,得到结构型模式实例集合、创建型模式实例候选集合以及行为型模式实例候选集合。

模式的形式化内容包括:a)每个模式有哪些参与者,包括类、类的属性、类的方法等;b)每个参与者应该包含的特征,包括可见性、方法的参数和返回值、属性的类型等;c)参与者之间的关系等,包括类之间的关联、聚合、继承、实现关系,类和方

法之间的关系,类和属性之间的关系等。

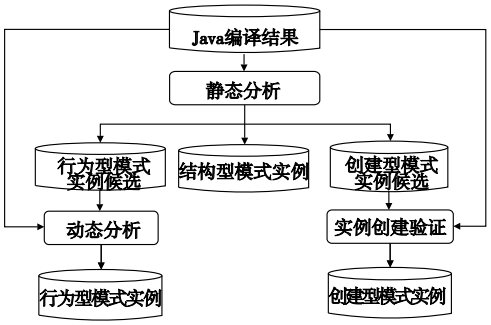

图1　静态分析和动态分析结合的设计模式抽取过程

源代码分析过程中,由于 Java 编译结果即 Java Bytecode 已经包含了足够的源代码的信息,因此本文使用了 ASM[1] 工具直接对编译后的结果进行分析。相对于从源代码中抽取来说,这样做的好处是抽取工作在没有源代码的情况下也能够进行。分析过程中去掉了与模式抽取无关的信息,最后形成 Java 代码的内部表示。

模式形式化结果与 Java 代码内部表示的匹配过程中,根据每个参与者的特征约束查找参与者实例候选集合,检查候选集合中每个模式的参与者关系约束和参与者实例之间的关系约束是否匹配,最后得到匹配结果。

由于基于静态结构分析的设计模式实例抽取方法已经在文献[12]中介绍,本文重点针对创建型及行为型模式实例的验证方法进行阐述。

2)实例创建验证　对于创建型模式候选实例来说,除匹配静态结构信息外还需要关心对象的创建情况和模式本身的特征是否匹配。因此对于创建型模式的候选,针对感兴趣的方法作数据流分析,获得对象的创建情况,作实例创建的验证;对于验证通过的候选,将其作为最终的创建型模式实例。

3)动态分析　对于行为型模式候选实例来说,除匹配静态结构信息外还需要关心运行期间对象之间的协作情况与模式本身的行为特征是否匹配。因此对于行为型模式的候选,为其设计测试用例并运行,运行的过程中获取程序的执行日志,然后将程序的执行日志与模式本身的行为特征进行匹配,对于匹配成功的候选,将其作为最终的行为型模式实例,这部分内容会在第 4 章进行详细描述。

3　数据流分析验证创建型模式实例

对于创建型模式来说,除了关注模式的结构以外,对某些对象的创建情况的分析也是非常重要的。如 Singleton 模式要求 Singleton 对象只能被创建一个、原型模式每次 prototype 操作也需要创建一个新的对象等。而简单地分析结构是无法获取这些信息的,因此对某些对象实例创建情况的分析能够对创建型模式进一步验证,提高识别精度。

本文通过为每个创建型模式建立对象创建的多重性约束,并且检查模式候选是否满足这些约束的方式来解决这个问题。a)对每个创建型模式建立多重性约束;b)根据这些约束对每个创建型模式候选进行数据流分析;c)将分析得到的结果与该模式的多重性约束相匹配,匹配通过的候选即为最终的结果,整个过程如图 2 所示。

3.1　建立多重性约束

在建立多重性约束的过程中,主要关心在指定的函数里面是否创建了感兴趣的对象以及创建的数量范围等。如 factoryMethod 模式需要在 factoryMethod( )函数里面创建 Product 对象(或者初始化时创建,在 factoryMethod( )方法里面返回创建好的对象),如果在该方法中并没有创建 Product 对象,则可能是软件工程师只是用来作为某个属性的 get 方法,并没有将其设计成为 Factory Method 模式。

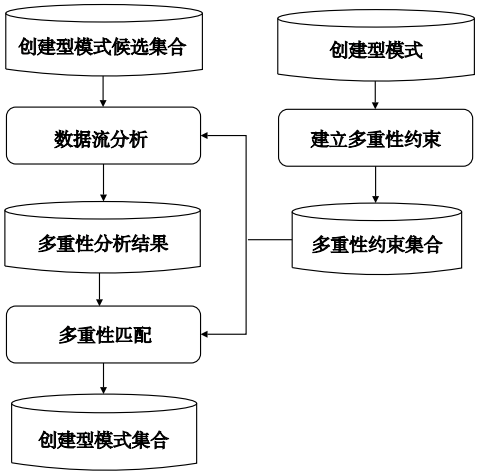

图2　创建型模式候选实例的验证过程

基于以上分析,本文将函数创建某个对象的数量范围作为主要的约束手段。

在描述这些约束之前,首先引入如下定义:

定义 1　multi{min, max}表示代码段创建实例的多重性,min 和 max 分别代表最小和最大可能的实例数量,由此定义多重性的两个运算:

a)select(multi1,multi2):选择运算,即两个处于选择执行关系的代码段创建实例的多重性,结果为 multi{min(multi1.min,multi2.min),max(multi1.max, multi2.max)}。

b)sequence(multi1,multi2):顺序运算,即两个处于顺序执行关系的代码段创建实例的多重性,结果为 multi{multi1.min + multi2.min, multi1.max + multi2.max}。

定义 2　multi_m(method × class)为某个方法中创建某个类型实例的多重性。

定义 3　multi_c(class × class)为初始化期间(考虑所有的初始化构造函数)创建某个类型实例的多重性。将所有初始化构造函数的多重性作选择运算,具体计算方法为

```
multi = {0,0}
for each c in constructors
  multi = select(multi,multi_m(c × class))
```

定义 4　multi_mc(method × class)为对象初始化到一个方法调用期间,创建某个类型实例的多重性。初始化期间的实例多重性与该方法的实例创建多重性作顺序运算,即 sequence(multi_m(method × class),multi_c(class × class))。

根据上述定义,本文以实例创建的多重性为关注点,将设计模式的多重性约束表示成以这些定义为基础的布尔表达式,并且为了方便后续的数据流分析,计算这些布尔表达式中所关注的函数集合。

下面以 Singleton 模式为例(结构如图 3 所示),说明设计模式多重性约束的建立方法。

Singleton 模式的关注点为 Singleton 实例的创建情况,有两种常见的实现方式:

a)在初始化时就创建 Singleton 实例(多重性为{1,1}),在 getInstance()方法里面简单地返回已经创建好的实例(多重性为{0,0})。因此可以表示为 multi_c(Singleton,Singleton) = {1,1} ∧ multi_m(getInstance(),Singleton) = {0,0}。

b)采用"Lazy"的方式创建 Singleton 实例,即在初始化时并不创建(多重性为{0,0}),而是在 getInstance()方法里面根据条件决定是否创建 Singleton(多重性为{0,1})。因此可以表示为 multi_c(Singleton,Singleton) = {0,0} ∧ multi_m(getInstance(),Singleton) = {0,1}。

基于上述两种实现方式,Singleton 模式的约束表达式为:

(multi_c(Singleton,Singleton) = {1,1} ∧ multi_m(getInstance(), Singleton) = {0,0}) ∨ (multi_c(Singleton,Singleton) = {0,0} ∧ multi_m(getInstance(),Singleton) = {0,1})

根据该约束表达式不难得出,关注的函数集合为{Singleton() × Singleton;getInstance() × Singleton}

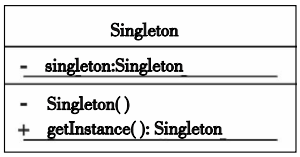

图3 Singleton模式结构

3.2 数据流分析

在数据流分析阶段,分析每个创建型模式,获取关注的函数集合,对于集合中的每个元素,从模式候选中查找匹配的函数,加入待分析列表,最后对待分析列表中的每个元素作数据流分析,得到分析结果。因此,数据流分析的关键在于获取某个方法创建感兴趣对象的数量范围,分析的过程中有如下几种情况:

a)简单语句或表达式。结果为创建对象的数量(使用 new 操作的数量),如果遇到函数调用,需要对调用的函数多重性进行分析,基于性能上的考虑,本文只分析了一层调用关系。

b)顺序语句。顺序执行的语句结果为两条语句的多重性结果作 sequence 操作。

c)选择语句。由于两条语句只有一条会被最终执行,因此结果为两条语句的多重性结果作 select 操作。

d)循环语句。由于循环体内的语句本身可能被循环 0 次或者多次,如果循环体内语句的多重性大于 0,则结果为{0,无穷大};否则结果为{0,0}。

具体算法如下:

多重性计算算法

```
multi(min,max)
multi function methodCreateObj(Method:m,Class:obj)
begin
    multi = statementCreateObj(m,block,obj)
end
multi function statementCreateObj(Statement:s,Class obj)
begin
    cass SEQUENCE:
      multi = (0,0)
      for each sub_stat in s
      begin
          multi = sequence(multi, statementCreateObj)(sub_stat, Obj))
end
    case IF-ELSE:
    multi = select(statementCreateObj)(s,ifBlock,obj),
        statementCreateObj(s,elseBlock,obj)
    case LOOP:
        multi. min = (0, statementCreateObj(s. block, obj)) * UNBUND)
    case SINGLE_STATEMENT:
    multi = {number of (call new obj)},number of(call new obj)
end
```

3.3 多重性匹配

在多重性匹配阶段,上步对模式候选的数据流分析的结果作为输入,判断其是否满足模式的多重性约束表达式,得到最终的模式实例集合。本文将模式的多重性约束记为 A,计算出来的多重性为 B,实际的检查结果有如下几种情况:

a)$A.\min \geq B.\min$ 且 $A.\max \geq B.\max$,即表达式的约束范围要比实际的检查结果大,判断结果为 true。

b)$A.\min \geq B.\min$ 且 $A.\max < B.\max$,即两个范围是有交叉的,并非包含关系。出现这样的情况有两种可能:(a)该候选并不是模式实例;(b)该候选是模式实例,只是由于多重性计算是基于静态分析的,因此可能要比运行期间可能创建的范围大(如软件工程师使用循环语句创建了有限数量的对象),导致出现范围交叉。此时工具不能判断属于哪一种情况,需要交给软件工程师判断是否接受模式候选。从抽取工具的结果分析情况来看,需要人工进行判断的候选大约占总体候选的 1%。

c)其他情况,判断结果为 false。

根据以上的原则,就可以计算出约束表达式的结果,对于最终判断结果为 true 的模式候选,并将其加入创建型模式实例集合。

4 动态分析验证行为型模式实例

对于行为型模式来说,除了静态结构以外,程序在运行时期每个对象之间的协作关系是否符合模式的特征也是需要验证的。因此需要运行程序并获取动态信息,将其与模式的动态特征相匹配,对行为型模式实例候选进行验证。

整个动态分析的过程大致分为以下三个步骤:

a)获取程序执行日志。对 Java bytecode(Java 代码编译的结果)进行 Instrument 处理,插入用于跟踪方法执行情况的代码,然后运行程序,获取程序的执行日志。

b) 行为模式的动态信息形式化。描述 GOF[4] 模式或者其他模式的动态行为,然后按照一定的规则将动态交互信息转换为状态图,最后将状态图形式化为 XMI 的格式表示。

c)动态行为匹配。将程序执行日志和模式动态行为形式化结果进行匹配,匹配成功的候选即为最终的结果。验证过程如图 4 所示。

4.1 获取程序执行日志

程序执行日志反映了程序的执行情况,包括能够反映对象之间交互细节的方法调用关系等。为了描述这些信息,本文使用了如下定义的数据结构。

定义 5 方法调用树(MCT)。测试用例的执行会形成一个方法调用关系,将每个执行过的方法作为一个节点,将该方法对其他方法的调用作为其子节点,这样形成的树称之为方法调用树(method-called tree,MCT)。树的根节点为测试用例执行的 main()方法。

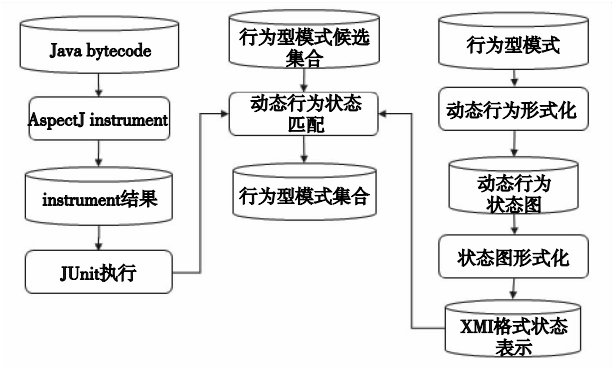

图4　行为型模式的动态分析过程

通过 MCT 的生成过程不难得出以下结论:如果节点 m_2 出现在以 m_1 为根节点的子树上,那么 m_1 中存在直接或者间接对 m_2 的调用。在判断两个方法之间是否存在调用关系时,本文将使用这个结论作为判定依据。

利用 AOP 工具 AspectJ[13],本文将 Java bytecode 进行 Instrument 处理,在关注的方法的入口和出口插入关注点,记录方法的签名。然后根据程序实现逻辑编写 JUnit[14] 测试用例,运行 Instrument 处理过的结果获取程序的执行日志,每一个用例的执行按照深度优先的顺序生成一个 MCT。

测试用例的选取需要尽可能地完备,测试场景遗漏可能会导致感兴趣的对象交互细节没有被抽取出来,从而影响动态验证结果(由于没有找到必要的交互细节造成识别的遗漏),这也是动态分析方法的局限性之一,文献[10,11]均受到了这个局限性的影响。

4.2　行为模式的动态信息形式化

对于大多数行为型模式来说,其动态行为以序列图的方式描述,但是对于少数模式,如职责链模式。参与的对象的个数、相应请求到哪里终止都是不确定的,这种情况下使用序列图很难准确描述。因此动态行为模式形式化主要解决两个问题:a)如何将序列图形式化;b)对于难以使用序列图进行准确描述的动态行为,如何进行描述。

由于 UML 状态图对参与对象的数量是不敏感的,本文引入 UML 状态图来描述模式的动态行为,而对于使用序列图描述的模式,本文按照一定的规则将序列图转换成为状态图。最后将状态图形式化为 XMI 格式的文档。下面分别讨论状态图的模型以及序列图到状态图的转换方法。

4.2.1　状态图的模型

标准 UML 状态图模型比较复杂,且一些描述手段在描述动态行为方面没有帮助,因此本文在标准的状态图的基础上进行了裁剪,裁剪之后的模型如图 5 所示。

每个模式的行为描述成为一个或多个状态图,每个状态图包含有一个初始状态、若干个中间状态和一个接收状态。另外还存在一个匹配上下文、用于保存状态匹配过程的信息。有时匹配过程会只关心 MCT 上某一个子树,而并不关心其他的节点,本文将匹配过程中正在关注的子树称之为匹配树。匹配树初始化为 MCT,随着后续的匹配操作可能会进行切换为 MCT 的某一个子树。

进入每个状态后可以进行一系列针对于上下文操作,如切换匹配树、从上下文中存取数据等。状态图中的状态转换由事件触发,事件主要为方法调用,但对于方法调用关系来说,软件工程师在实际编写代码时,可能使用直接方式调用,在 MCT 上体现为调用者和被调用者是父子关系;也可能采用间接的方式调用,在 MCT 上表现为被调用者是以调用者为根的子树上的某一个节点。基于以上考虑,本文使用 exists 函数来描述两个方法之间的调用关系。所以事件表示成关于方法调用的布尔表达式。

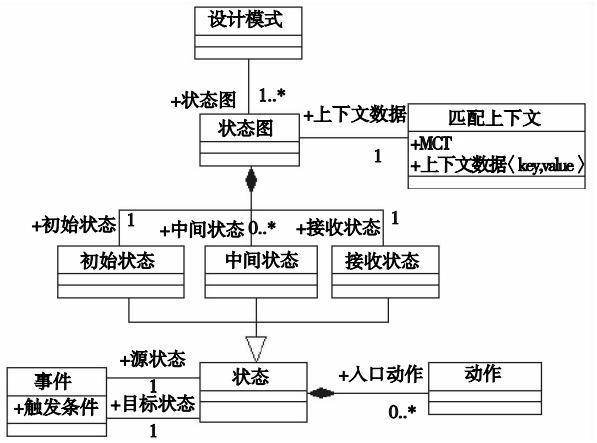

图5　状态图模型

为了不局限于特定的模式,本文提供了如下可以在事件和动作描述中使用的函数:

a) child(node × index)node。取 node 节点的第 index 个子节点。

b) parent(node)node。取 node 节点的父节点。

c) switch_tree(node)void。切换匹配树到以 node 为根节点的子树上。

d) current()node。取当前正在关注的节点,用于获取满足 exists()函数条件的节点。

e) root()node。取当前匹配树的根节点。

f) any(method)set〈method〉。表示 method 本身以及重写(子类重写父类)method 的集合。

g) exists(method1,method2)bool。判断当前匹配树中是否存在 method1 对 method2 的调用。具体的方法是查看 method2 所在的节点是否存在于以 method1 为根节点的子树上。如果存在则将符合条件的 method2 节点放入当前关注的节点中,并返回 true,否则返回 false。

h) put_data(key,value)void。向匹配上下文放数据。

i) get_data(key)value。从匹配上下文中取数据。

4.2.2　序列图到状态图的转换

关于把序列图转换成状态图需要处理很多细节的问题,比如如何处理各种消息类型、对于多个序列图是否需要合并以及如何合并等。由于篇幅的限制,本文只针对单个序列图的几种事件流的转换方法加以描述。

序列图中的事件流分为顺序执行的事件流、有条件分支的事件流和循环的事件流三种情况,下面分别加以讨论:

a)顺序事件流。定义一个状态图的初始状态 t_0,对于序列图中的每个消息,加入一个新状态 t_i,并加入由状态 t_{i-1} 到状态 t_i 的转换,转换的条件是存在消息所在的方法调用关系。如图 6 所示。

b)分支事件流。将分支事件流中的顺序事件流进行转

换,假设入口状态和出口状态为 t_i 和 t_j,如果 t_{j+1}存在,则加入 t_{i-1}到 t_{j+1}的转换关系,条件为 t_j 到 t_{j+1}的转换条件;否则加入 t_{i-1}到接收状态的转换关系,条件为 ε,如图 7 所示。

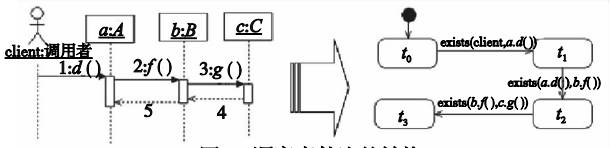

图6 顺序事件流的转换

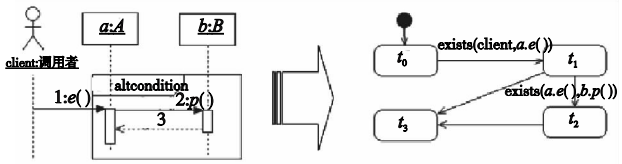

图7 分支事件流的转换

c)循环事件流。在顺序事件流的转换结果基础上,加入 t_j 到 t_i 的转换,条件为 ε,如图 8 所示。

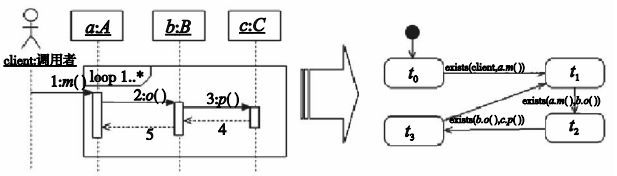

图8 循环事件流的转换

4.3 动态行为状态匹配

根据行为型模式的候选,将获取到的程序执行日志与形式化结果进行匹配,匹配最终如果能够到达状态图的接收状态,则认为当前场景满足动态行为约束。如果一个模式的所有状态图都曾经到达过接收状态,则将该模式候选添加到最终的行为型模式集合中。匹配的具体算法如下:

a)状态图实例的创建。计算每个状态图的入口条件,对于满足入口条件的状态图,创建一个状态图实例。

b)状态图实例创建首先生成状态图上下文实例,将 MCT 作为状态图的初始匹配树,计算每个状态的出口条件,开始后续的匹配。

c)到达一个新状态后执行该状态下的动作,然后计算该状态出口条件是否匹配,匹配成功的条件进行状态切换。如果一个状态的多个出口条件都为 true,则以到达接收状态的条件优先级最高、出口条件为空且未到达接收状态的条件优先级最低为原则。

d)如果一个状态的所有出口条件为 false,并且未达到接收状态,则匹配失败。

对于同一个状态图,还需要处理多次匹配同一个 MCT 即同一个 MCT 的匹配可能会创建多个状态图的实例的情况,以提高匹配精度。

4.4 职责链模式的例子

职责链模式的结构如图 9 所示。

行为形式化的结果如图 10 所示。

形式化说明如下:

Init_state:初始状态,不需要条件可以进入该状态。

State_1:当 MCT 中存在 client 对任何一个 handleRequest( )调用时,则开始匹配。由于本文接下来需要关注 handleRequest( )是否将处理动作委托给 sucessor,因此需要进行匹配树的切换,将 handleRequest( )作为匹配树的根节点。

State_2:当前的匹配树存在 handleRequest( )对 succssor 调用时(目前的职责链上面至少有两个参与者了),则切换匹配树,继续匹配。

结束状态:在调用 handleRequest( )的过程中,至少有两个以上的参与者参与,作为判定标准。

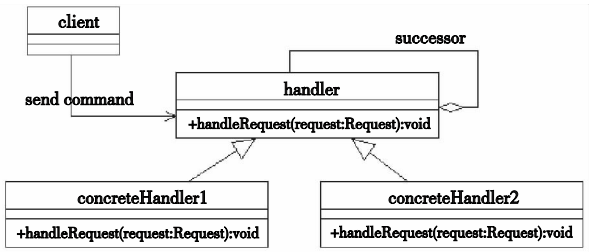

图9 职责链模式结构

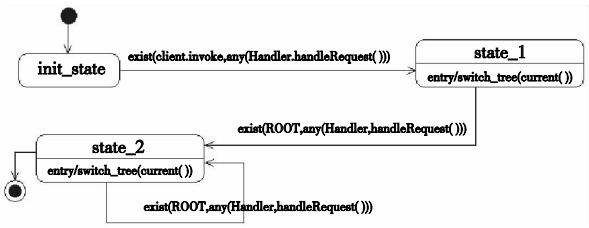

图10 职责链模式的行为形式化结果

下面以两个场景为例说明匹配过程。

场景 1 ConcreteHandler1 处理完毕,并没有将请求转发给 ConcreteHandler2,MCT 如图 11 所示。

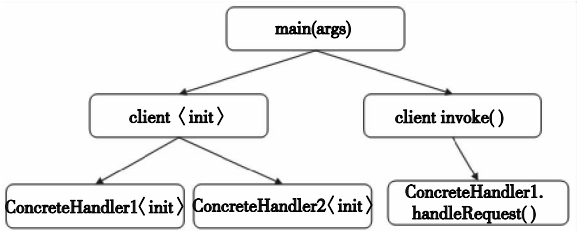

图11 职责链模式验证场景1

匹配的结果(未到达接收状态)如图 12 所示。

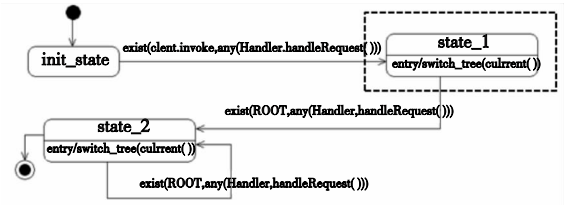

图12 对应场景1的匹配结果

场景 2 ConcreteHandler1 未处理完毕,交给 ConcreteHandler2 处理,由 ConcreteHanlder2. handle Request( )完成处理,MCT 如图 13 所示。

匹配的结果(到达接收状态)如图 14 所示。

5 实验结果

为了对抽取的算法进行验证,本文进一步完善了文献[15]中的设计模式抽取工具 DPDT,并选取了开源工具 Jhotdraw 5.1 进行分析,抽取其中存在的 factory method、observer、state/strategy 模式实例,并将抽取的结果与 DEMIMA[10]、Tsantalis 等人方法[2]、DPRE[11]等几个工具作了比较,结果如表 1、2 所示。

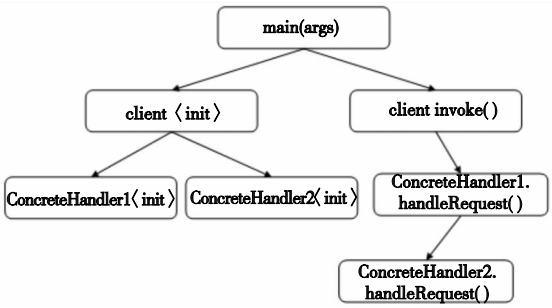

图13　职责链模式验证场景2

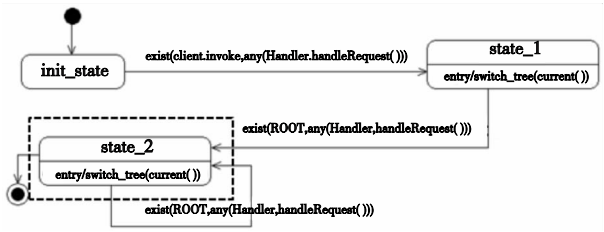

图14　对应场景2的匹配结果

表 1　各个工具抽取实例数量比较

工具	factory method			observer			state/strategy		
	TP	*FP*	*T*	*TP*	*FP*	*T*	*TP*	*FP*	*T*
DEMIMA	3	186		2	5		6	15	
Tsantalis 等人方法	2	0	3	1	4	5	20	2	23
DPRE	/	/		5	4		21	57	
DPDT	3	2		5	3		15	22	

表 2　各个工具的回溯率和准确率比较

工具	factory method		observer		state/strategy	
	RC/%	*PR*/%	*RC*/%	*PR*/%	*RC*/%	*PR*/%
DEMIMA	100	1.6	40	29	26	29
Tsantalis 等人方法	66.7	100	20	20	87	91
DPRE	/	/	100	56	91	27
DPDT	100	60	100	63	65	41

这里使用 *TP* 表示工具抽取的为真的实例数,*FP* 表示工具抽取的为假的实例数,*T* 表示通过人工查看软件文档或源代码的方式找出的实例数。回溯率定义为 recall = *TP*/*T*,作为评价抽取实例遗漏情况的标准,记为 *RC*;准确率定义为 precision = *TP*/(*TP* + *FP*),作为评价抽取准确程度的标准,记为 *PR*。

从结果上看:

a)在创建性模式识别方面 DPDT 由于加入了实例数量验证的方法,相对 DEMIMA[10]、Tsantalis 等人方法[2]工具来说,precision 和 recall 分别具有比较明显的优势;DPRE[11]工具侧重于行为型模式的抽取,对于创建型模式的抽取并未提及。工具的缺点是少部分实例的识别需要人工来判定。

b)在行为型模式识别方面,对于 observer 的抽取,recall 和 precision 都比较高;而对于 state/strategy 的抽取,和 Tsantalis 等人方法[2]工具相比还存在一定的差距。

6　结束语

从源码中抽取设计模式对于提高软件可理解性、可维护性及软件演化具有重要意义。本文面向 Java 语言提出了静态分析和动态分析相结合的源代码设计模式的抽取方法,在静态分析得到模式候选实例后,分别使用数据流分析和动态分析对创建型模式实例和行为型模型实例进行验证,并深入讨论了方法的可行性,进一步提高了源码中设计模式抽取的准确率。

本文需要进一步研究的工作有:a)多重性匹配过程中一些场景需要软件工程师来介入,如何降低甚至消除这样的场景有待进一步研究;b)测试用例的选取一定程度上影响动态行为匹配的准确性,测试用例选择不够充分的情况下,一些行为模式会因为状态图没有被创建或者没有匹配成功而造成识别遗漏,因此需要考虑结合覆盖率工具检查等方式提高测试用例的覆盖程度,进一步提高匹配的精度;c)模式变体的存在也给查找的准确度和匹配的精度带来一定的困难,因此针对模式变体的静态分析和动态分析方法也是需要进一步研究的对象。

参考文献:

[1] ASM[EB/OL]. http://asm.ow2.org/.

[2] TSANTALIS N, CHATZIGORGIOU A, STEPHANIDES G, *et al*. Design pattern detection using similarity scoring[J]. IEEE Trans on Software Engineering, 2006, 32(11): 896-909.

[3] SHI N, OLSSON R. Reverse engineering of design patterns from Java source code[C]//Proc of International Conference on Automated Software Engineering. 2006: 123-134.

[4] GAMMA E, HELM R, JOHNSON R, *et al*. Design patterns: elements of reusable object-oriented software[M]. Menlo Park: Addison Wesley, 1995.

[5] WENDEHALS L. Improving design pattern instance recognition by dynamic analysis[C]//Proc of ICSE Workshop on Dynamic Analysis. 2003: 29-32.

[6] WENDEHALS L, ORSO A. Recognizing behavioral patterns at runtime using finite automata[C]//Proc of Workshop on Dynamic Systems Analysis. 2006: 33-40.

[7] Von DETTEN M, PLATENIUS M C. Improving dynamic design pattern detection in reclipse with set objects[C]//Proc of the 7th International Fujaba Days. 2009: 15-19.

[8] PETTERSSON N. Measuring precision for static and dynamic design pattern recognition as a function of coverage[C]//Proc of Workshop on Dynamic Analysis. 2005: 43-49.

[9] NG K Y J, GUÉHÉNEUC Y G. Identification of behavioral and creational design patterns through dynamic analysis[C]//Proc of Workshop on Program Comprehension through Dynamic Analysis. 2007: 34-42.

[10] GUEHENEUC Y G, ANTONIOL G. DEMIMA: a multi-layered approach for design pattern identification[J]. IEEE Trans on Software Engineering, 2008, 34(5): 667-684.

[11] LUCIA A D, DEUFEMIA V, GRAVINO C, *et al*. Improving behavioral design pattern detection through model checking[C]//Proc of the 14th European Conference on Software Maintenance and Reengineering. 2010: 176-185.

[12] De LUCIA A, DEUFEMIA V, GRAVINO C, *et al*. Design pattern recovery through visual language parsing and source code analysis[J]. Journal of Systems & Software, 2009, 18(7): 1177-1193.

[13] AspectJ: a seamless aspect-oriented extension to the Java programming language[EB/OL]. http://www.eclipse.org/aspectj/.

[14] JUnit [EB/OL]. http://www.junit.org/home.

[15] 冯铁,李文锦,张家晨. 面向 Java 语言的设计模式抽取方法的研究[J]. 计算机工程与应用, 2005, 41(25): 28-33.