

一种基于横切特征分析的软件体系结构自动重构方法

黎丙祥 沈立炜 彭鑫 赵文耘

(复旦大学计算机科学技术学院 上海 200433)

摘要 软件体系结构中的横切关注点增加了软件体系结构的复杂性,从而加剧了体系结构演化与维护的困难。这种设计问题可以通过体系结构层面的重构来进行改善。在已有的横切特征分析方法基础上,提出了一种面向横切特征分析的体系结构自动重构方法。该方法首先基于特征与构件之间的追踪关系分析横切特征,然后将与横切特征有直接追踪关系的构件从初始体系结构中提取出来,实现方面构件,完成体系结构重构。在面向方面体系结构描述语言 AO-ADL 基础上开发了相应的体系结构重构工具,并针对一个业务系统进行了体系结构重构实验。实验结果表明,该方法能有效地实现体系结构横切特征的自动化重构。

关键词 特征,横切,行为,方面绑定,方面构件,体系结构,重构

Crosscutting Feature Analysis-based Automatic Software Architecture Refactoring Method

LI Bing-xiang SHEN Li-wei PENG Xin ZHAO Wen-yun

(College of Computer Science and Technology, Fudan University, Shanghai 200433, China)

Abstract Crosscutting concerns in software architecture increase the complexity of software architecture and the difficulty for evolution and maintenance. This design problem can be improved by refactoring on architectural level. This paper offered an automatic software architecture refactoring method based on the existing analysis of crosscutting feature. At first, this method analyzes crosscutting features based on traceability between features and components. Then those components which are direct trace relation to these crosscutting features are extracted from initial architecture as aspectual components, finishing architecture refactoring. We developed a architecture refactoring tool based on Aspect-Oriented Architecture Description Language AO-ADL. Experiments on architecture refactoring were taken on a business system. The result shows this method can refactor crosscutting concerns in software architecture effectively and automatically.

Keywords Feature, Crosscutting, Behaviour, Aspectual binding, Aspectual component, Architecture, Refactoring

1 引言

特征用来表示系统中用户/客户可见的需求^[1]。特征模型用于软件需求建模,定义了特征及它们之间的约束关系^[9,10]。特征模型中存在影响多个其它特征的横切特征,导致软件系统复杂性以及演化与维护困难的增加。在软件开发中,尽早发现这种横切关注点并加以适当处理,可以降低开发的难度,同时可以相应地提高软件质量^[4]。因此,发现特征模型中的横切特征并在体系结构设计时加以考虑,进而在实现级利用 AOP 等方面实现语言加以实现,对于改善系统设计及实现的质量具有重要的作用。

软件体系结构描述了一个软件系统构成要素的总体组织结构,包括计算单元及它们之间的通信^[2]。一般来说,体系结构定义了构件与连接器之间的关系,构件封装了计算,连接器封装了构件之间的通信和交互逻辑。也就是说,体系结构分

离了计算与交互^[11]。然而,除了封装在单个构件和连接器中的计算及交互逻辑之外,体系结构中还可能存在着横切多个体系结构单元的横切单元。这些横切单元往往与某些非功能性关注点相关,例如服务于安全性目标的日志记录、权限检查、加密/解密等。针对这一问题,文献^[3]提出了一种面向方面的体系结构描述语言 AO-ADL,为体系结构中横切行为描述提供了支持。

体系结构中的这种横切特性大多数情况下是有害的。首先,实现横切行为的构件使得体系结构中构件之间的连接关系更加复杂,不利于体系结构演化与维护。其次,这一问题将导致实现层代码的混杂和散布,增加了代码的长度并且影响代码的模块化程度。此外,构件中所混杂的横切特性降低了构件本身的内聚度和可复用性,例如订单处理构件如果包含权限检查等与特定应用密切相关的横切特性,将很难在其它应用中进行复用。

到稿日期:2009-10-13 返修日期:2010-01-11 本文受国家自然科学基金(No. 60703092)以及国家 863 计划(No. 2007AA01Z125, No. 2009AA010307)资助。

黎丙祥(1984-),男,硕士生,主要研究方向为软件产品线,E-mail:072021149@fudan.edu.cn;沈立炜(1982-),男,博士生,主要研究方向为领域工程、软件产品线,E-mail:061021062@fudan.edu.cn(通讯作者);彭鑫(1979-),男,博士,讲师,CCF 会员,主要研究方向为构件技术与软件体系结构、软件产品线、软件再工程;赵文耘(1964-),男,硕士,教授,博士生导师,CCF 高级会员,主要研究方向为软件复用及构件技术。

由此可见,分析特征模型中的横切特征,并根据这些横切特征对体系结构进行方面化重构是有意义的。目前已经有了一些基于需求和体系结构模型的早期方面分析方法(例如文献[4]),但对于相应的重构方法的研究工作还相对较少。本文在文献[4]提出的体系结构横切特征分析方法的基础上,对面向横切特征的体系结构重构方法进行了研究。我们的方法扩展了文献[4]中介绍的横切分析方法,将特征与构件之间的追踪关系分为直接追踪关系与间接追踪关系,首先根据特征与体系结构层构件之间的追踪关系,分析特征模型中的横切特征。然后,根据这些横切特征将初始体系结构中中与这些横切特征有直接追踪关系的构件提取出来,表示为方面构件,实现自动化重构,重构后的体系结构能够体现这些横切行为。在此方法的基础上,我们针对 AO-ADL 描述的体系结构开发了横切分析与体系结构重构工具原型,并对一个运输管理业务系统进行了实验。初步实验结果表明,该方法能找出横切特征,并能根据这些横切特征对体系结构进行重构。

本文第 2 节介绍横切分析模式与面向方面的体系结构描述语言 AO-ADL;第 3 节详细介绍横切特征分析与体系结构重构;第 4 节介绍基于 AO-ADL 的体系结构重构工具原型,另外对运输管理系统做体系结构重构的实验;第 5 节结合实验结果,对本文方法的优缺点以及以后的改进方向进行了探讨;最后对全文进行总结。

2 相关工作

2.1 横切模式

在横切分析方面,Klass van den Berg 等人^[4]提出了基于追踪性的软件开发早期横切分析框架。追踪性就是软件开发过程中各阶段软件制品之间的依赖关系(比如用况与构件之间),这种依赖关系通常用追踪矩阵表示。在面向方面的软件开发中,横切通常是根据散布(scatter)与交织(tangle)来描述的。文献[4]给出了横切的形式化定义。设 Source 与 Target 分别是软件开发不同阶段软件制品的集合,从 Source 到 Target 存在追踪关系。

在 Source 到 Target 的映射 f 中,Source 中的元素 s 在 Target 中有多个元素与之对应时,就说 s 发生散布^[4]。形式化定义如下: s 发生散布,当且仅当 $s \in \text{Source}, f(s) = \{t \in \text{Target} | t \text{ 与 } s \text{ 是相关的}\}$,且 $\text{Card}(f(s)) > 1$,Card 表示集合的势。

在 Source 到 Target 的映射 f 中,Source 中的多个元素对应到 Target 中的同一元素 t ,就说 t 发生交织^[4]。形式化定义如下: t 发生交织,当且仅当 $t \in \text{Target}$,且 $\text{Card}(f^{-1}(t)) > 1$ 。

在 Source 到 Target 的映射 f 中,Source 中的一个元素 s 散布到 Target 中的多个元素,并且在这些元素中至少存在一个发生了交织,且其中包含了 s ,就说 s 发生了横切^[4]。形式化定义如下: $s_1, s_2 \in \text{Source}$ 且 $s_1 \neq s_2, s_1$ 横切 s_2 当且仅当 $\text{Card}(f(s_1)) > 1$,且 $t \in f(s_1), \text{Card}(f^{-1}(t)) > 1, s_2 \in f^{-1}(t)$ 。

举例说明:Source = { s_1, s_2, s_3, s_4 }, Target = { t_1, t_2, t_3, t_4, t_5 }。假设 Source 是用况的集合,而 Target 是构件的结合。已知 $f(s_1) = \{t_1, t_3, t_4\}, f(s_2) = \{t_2\}, f(s_3) = \{t_3\}, f(s_4) = \{t_5\}$,上面定义得出 s_1 发生了散布, t_3 发生了交织, s_1 横切 s_3 。

2.2 AO-ADL

Monica Pinto 等人^[3]提出了一种基于 XML 的面向方面的体系结构描述语言 AO-ADL。与传统 ADL 类似,AO-ADL 也支持构件与连接器的行为协议描述。构件行为描述表示构件接口中方法的并发顺序,AO-ADL 中,这些行为描述是可扩展的。AO-ADL 中构件与连接器是体系结构的基本元素,其大致结构见图 1。AO-ADL 通过扩展连接器语义来支持体系结构上横切行为的描述,连接器描述增加了方面角色与方面绑定,目的是支持横切行为描述。方面角色表示与之连接的构件是方面构件,方面构件表示横切关注点,其中构件是否是方面构件主要看其与连接器之间的连接关系。方面绑定包括切入点描述、Advice、Operator 与 Advice Ordering 等。Operator 指 Advice 绑定是 Before, After 还是 Around。Advice Ordering 描述了 advice 的执行顺序。

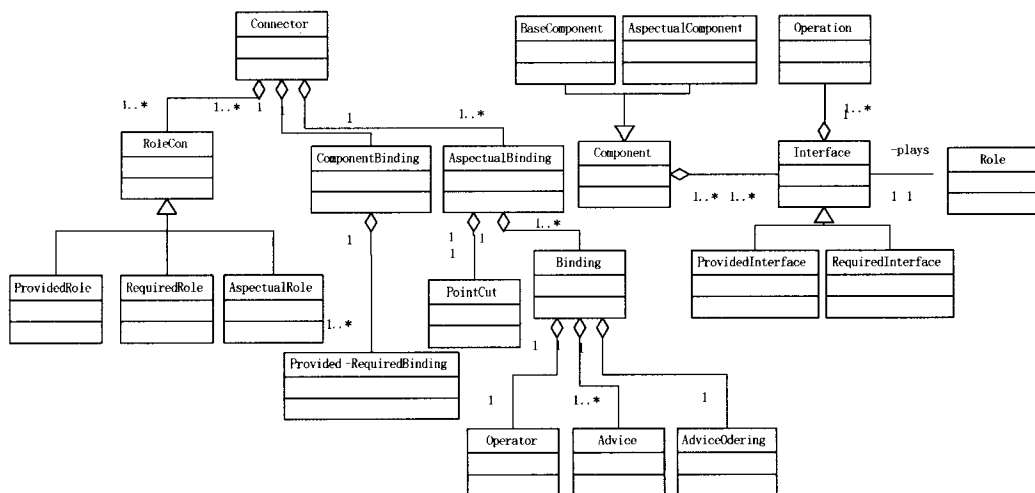


图 1 AO-ADL 基本结构^[3]

2.3 构件行为协议

Robert Allen 等人^[7,8]提出了一种基于 CSP 的体系结构形式化描述语言 Wright。构件描述包括接口与计算的描述。其中计算描述由接口中的方法序列组成,体现了各接口中方法

的并发顺序。由构件计算描述我们可以知道构件接口中方法是如何组合完成构件功能的。

Wright 中的构件计算描述与 AO-ADL 语言中的构件行为描述是一致的。我们采用 Wright 来描述 AO-ADL 中构件

的行为,这样做有以下几个方面的原因:首先, AO-ADL 语言中构件的行为描述本身是可扩展的,并不限于哪种语言描述;其次, Wright 有强大的形式化描述能力,能很好地描述构件的行为(包括顺序、选择等),并且这种形式化描述便于解析;最后, Wright 提供的构件并发性描述是体系结构重构中方面构件提取的基础。提取方面构件时的连接点、切入点和通知来自于对行为协议的解析。关于这一点,本文后面再做详细说明。

本文将借鉴文献[4]提出的基于追踪关系的横切分析模式,利用特征模型中特征与体系结构中构件之间的追踪关系来分析横切特征,然后根据分析结果对体系结构进行方面化重构。目前为止还没出现体系结构上横切行为方面化重构的研究,因此这种研究是有意义的。

3 横切特征分析与体系结构重构

3.1 方法概述

本文的方法在扩展的特征构件追踪关系的基础上,进行横切特征分析,得到横切特征,然后在体系结构层次上对这些横切特征的直接追踪关系的构件进行方面化重构。这里的重构就是将初始体系结构上的横切行为提取出来,用方面构件来表示。方法基本流程见图 2,包含两个关键步骤,分别介绍如下。

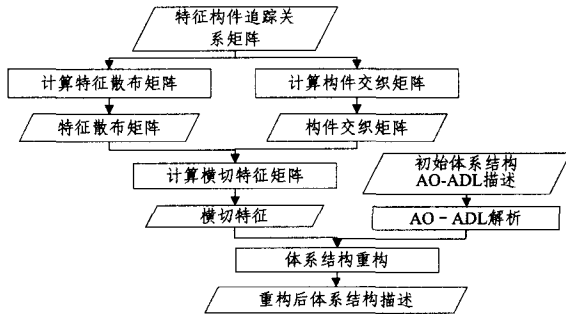


图 2 方法概述

第一步,横切特征分析。我们扩展了文献[4]中提出的横切分析方法,选取特征模型中的特征为 Source,体系结构中的构件为 Target。根据输入的特征构件追踪关系矩阵进行横切特征分析,得到横切特征矩阵。具体包括 3 个小步骤:

- 1) 计算特征散布矩阵;
- 2) 计算构件交织矩阵;
- 3) 计算横切特征矩阵。

第二步,体系结构重构。经过上一步得到横切特征后,在体系结构层次上将这些横切特征的直接追踪关系的构件以方面构件提取出来。

3.2 横切特征分析

这部分介绍分析扩展的特征构件追踪关系矩阵,得到横切特征矩阵的详细过程,并对实例进行分析。

3.2.1 横切特征分析方法

我们的方法之所以选择特征与构件进行横切分析,主要有以下几个原因:第一,在软件开发中,越早发现横切,就能越早在设计阶段对横切进行建模,进而可以在实现级将横切进行 AOP 实现。这样,在软件开发早期就将横切提取出来,提高了软件的模块化,便于演化与维护。第二,需求中的横切关注点就是横切了其它需求制品的关注点,体系结构中的横切

关注点就是横切了体系结构制品中的关注点^[6]。我们的目标是发现需求层中的横切关注点,即特征,然后指导体系结构中的横切行为方面化提取。第三,散布是单个软件需求散布在面向对象设计的多个类中,交织是设计中的单个类对应了多个软件需求^[5]。这为本文提出的横切特征分析方法提供了依据。

我们对文献[4]中的追踪关系矩阵进行了扩展,将特征与构件之间的追踪关系分为直接追踪关系与间接追踪关系。直接追踪关系表示构件是该特征的直接实现代码,比如在一个运输管理系统中,如果新建与关闭一个运输订单都要记录日志,则与一个日志记录 Log 特征有追踪关系的构件包括 Log 构件、新建订单 Create Order Process 构件、关闭订单 Close Order Process 构件,而 Log 构件是直接追踪关系的构件,其余构件为间接追踪关系的构件。另外,假定一个特定特征的直接追踪关系的构件只有一个,如果有多个的话,就当成一个复合构件处理。将追踪关系进行扩展的主要目的是便于体系结构的重构。

根据文献[4],不难得出横切特征分析算法。算法的输入是特征构件追踪关系矩阵 $traceability[n][m]$,代表 n 个特征与 m 个构件之间的追踪关系。输出为 $crosscutting[n][n]$,表示 n 个特征之间的横切关系。若 $crosscutting[i][j]=1$,表示特征 i 横切特征 j ;若对每个 $j, 0 \leq j < n, crosscutting[i][j]=0$,则特征 i 不是横切特征,否则为横切特征。显然,横切特征矩阵不一定是对称矩阵。具体算法见图 3。

Input: 特征构件追踪关系矩阵 $traceability[n][m]$, n 为特征数目, m 为构件数目

Variable: 特征散布矩阵 $scatter[n][m]$, 构件交织矩阵 $tangle[m][n]$, $temp[n][n]$, 存储 $scatter[n][m]$ 与 $tangle[m][n]$ 的乘积

Output: 特征横切关系矩阵 $crosscutting[n][n]$

for $i=0$ to $n-1$ //计算特征散布矩阵
begin

if 如果特征 i 发生散布 then
for $j=0$ to $m-1$
 $scatter[i][j]=traceability[i][j]$

else
for $j=0$ to $m-1$
 $scatter[i][j]=0$

end

for $j=0$ to $m-1$ //计算构件交织矩阵

begin

if 如果构件 j 发生交织 then
for $i=0$ to $n-1$
 $tangle[j][i]=traceability[i][j]$

else
for $i=0$ to $n-1$
 $tangle[j][i]=0$

end

$temp=matrix_multiply(scatter, tangle, n, m)$ //矩阵相乘

for $i=0$ to $n-1$

begin

for $j=0$ to $m-1$
begin
if $temp[i][j] > 0$ and $i \neq j$ then
 $crosscutting[i][j]=1$

else
 $crosscutting[i][j]=0$

end
end

图3 横切特征分析算法

3.2.2 实例分析

我们使用运输管理系统 (Transportation Management System, TMS) 进行横切特征分析。系统特征模型见图4, 初始体系结构见图5。该系统场景如下: Receiver 负责接受客户并创建运输订单, Scheduler 调度订单, 即为运输订单分配卡车运输。此后, 控制权交给 Manager, 由 Manager 负责完成任务。然后回收卡车, 以便以后分配。其中, 创建与调度订单之前要做权限检查, 创建与完成订单后记录 Log, 以便以后查询。

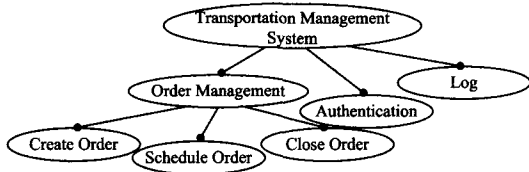


图4 特征模型

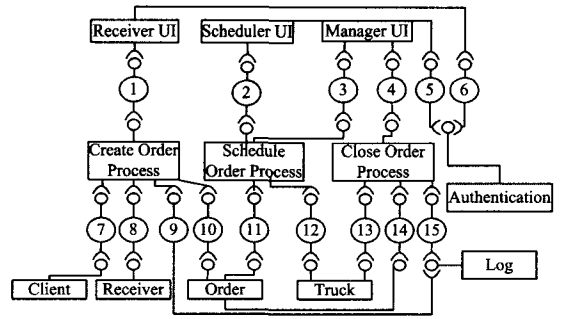


图5 初始体系结构

针对 TMS 系统, 我们给出了扩展的特征-构件追踪关系矩阵 traceability, 见表1。0 表示特征与构件之间不存在追踪关系, 1 表示存在追踪关系。其中, 灰色框表示直接追踪关系。我们运用 3.2 节的横切特征分析方法, 可以分别得出特征散布矩阵 scatter、构件交织矩阵 tangle 和特征横切关系矩阵 crosscutting。限于篇幅, 在此只给出 crossCutting 矩阵, 见表2。由表2可知, Log 与 Authentication 是横切特征, 需要进行重构。

表1 特征-构件追踪关系矩阵

Components	Receiver	Scheduler	Manager	Create Order	Schedule	Close Order	Log	Authentication	Client	Receiver	Order	Truck
Features	UI	UI	UI	Process	Order Process	Process						
Create Order	0	0	0	1	0	0	0	0	0	0	0	0
Schedule Order	0	0	0	0	1	0	0	0	0	0	0	0
Close Order	0	0	0	0	0	1	0	0	0	0	0	0
Log	1	0	1	0	0	1	1	0	1	0	0	1
Authentication	0	1	0	1	1	0	0	1	0	1	1	0

表2 横切特征矩阵

Features	Create Order	Schedule Order	Close Order	Log	Authentication
Create Order	0	0	0	0	0
Schedule Order	0	0	0	0	0
Close Order	0	0	0	0	0
Log	0	0	1	0	0
Authentication	1	1	0	0	0

3.3 体系结构重构

这部分详细介绍在第一步横切特征分析的基础上对初始体系结构进行方面化重构的过程, 也就是提取横切方面构件的过程。AO-ADL 对体系结构上横切行为描述提供了支持, 面向方面程序设计(AOP)提供了实现层的横切编程机制。

初始体系结构除了关注计算与通讯的分离外, 没有关注其它的横切行为, 这是不够的。原因如下: 第一, 横切的存在使得软件体系结构连接关系错综复杂, 难以演化与维护, 软件产品线体系结构还涉及到可变性问题, 这种复杂性更加体现出来; 第二, 体系结构为模块化提供了指导, 横切的存在明显违背了高内聚、低耦合的模块设计原则。

2.3 节提到的构件行为协议描述是体系结构重构中方面构件提取的基础。我们使用 Wright 体系结构描述语言的行为描述方法。方面构件提取方法见表3。表3给出了已知通知类型 (Advice Type)、构件行为 (Behaviour) 和通知 (Advice)、求连接点 (JoinPoints) 与重构后构件行为 (Behaviour_Refactoring) 的方法。其中 preMethods (behavior, method) 表示 Behaviour 中 method 的直接前缀方法集合; postMethods (behavior, method) 表示 Behaviour 中 method 的直接后缀方法集

合; postMethodsOfMethod (behavior, method) 表示 method 执行后且使得判断条件为真的后缀方法集合; preMethodsOfMethod (behavior, method) 表示 method 执行前的方法集合。

表3 方面构件提取方法

Advice Type	Behaviour	Advice	JoinPoints	Behaviour (重构后)
After	... -> methodA >method->...	method	methodA 等, 即 preMethodsOfMethod (behavior, method)	... -> methodA ->...
Before	...->method-> methodA->...	method	methodA 等, 即 postMethodsOfMethod (behavior, method)	... -> methodA ->...
Around	...->method-> (methodA methodB) ->...	method	methodA 等, 即 postMethodsOfMethod (behavior, method)	... -> methodA ->...
Around	...-> methodA -> (method II methodB) ->...	method	methodA 等, 即 preMethodsOfMethod (behavior, method)	... -> methodA ->...

根据 AOP, 将通知类型分为 Before, After, Around 3 种类型。其中 Around 机制更加丰富, 既可以在连接点之前执行, 也可以在连接点执行。我们将其细分为两种: 第一种是在连接点 (Join Point) 之前执行通知, 并根据通知执行情况判断连接点是否执行, 比如 method->(methodA || methodB), 这里约定 methodA 是 method 执行且条件为真才执行的方法; 第二种是执行连接点之后判断通知是否执行, 比如 methodA->(method || methodB)。

在方面构件提取方法的基础上, 我们提出了体系结构重构算法。算法的输入是初始体系结构的 AO-ADL 描述、第一

步横切特征分析得到的横切特征矩阵以及重构方面构件的横切类型(Advice Type);输出是重构后的体系结构 AO-ADL 描述文件。体系结构重构算法的关键步骤见图 6。

```

Input: crosscuttingfeatures[p], 横切特征集合
      traceability[n][m], 特征构件追踪关系矩阵
      initial, 初始体系结构, 解析体系结构 AO-ADL 描述得到
Variable: advicetype 通知类型, component, 构件, connector, 连接器
Output: result, 重构后体系结构
for i= 0 to p-1
begin
    得到与特征 i 有直接追踪关系的构件 component
    判断方面的通知类型 advicetype
    在体系结构 initial 中找到构件 component 的服务接口集合 providedInterfaces
    for j=0 to length(providedInterfaces)-1
    begin
        providedinterface=providedInterfaces[j]
        得到与服务接口 providedinterface 有连接关系的构件集合 basecomponents
        得到服务接口 providedinterface 中的方法 advicemethod
        for k=0 to length(basecomponents)-1
        begin
            temp=basecomponents[k]
            解析构件行为协议, 由表 3, 得到连接点方法集合 joinpointmethods
            搜索 temp 的所有接口, 找到包含 joinpointmethods 中方法(可以只包含一个)的接口集合 interfaces
            for e=0 to length(interfaces)-1
            begin
                interface=interface[e]
                得到接口 interface 包含在 joinpointmethods 中的方法集合 joinpoints,
                找到与 interface 接口相连的所有连接器集合 connectors
                for m=0 to length(connectors)-1
                begin
                    connector=connectors[m]
                    创建一个方面角色 aspectualrole, 对应接口为 providedinterface, 加到 connector 中
                    创建一个切入点 pointcut, 设置其连接点为 joinpoints
                    创建一个方面绑定 aspectbind, 设置切入点 pointcut, 通知类型 advicetype, 添加 advicemethod 为 connector 增加一个方面绑定 aspectbind
                end
            end
        end
    end
    到此, 构件 temp 的 interfaces 中所有接口都重构完成, 由表 3 更新 temp 的行为协议 behaviour, 去掉构件 temp 与服务接口 providedinterface 对应的请求接口及之间的连接器
end
end
end
result=initial //所有操作都在 initial 上进行, 重构结束后 initial 即是 result
    
```

图 6 体系结构重构算法

为了实现基于横切特征的体系结构重构,用 Java 开发了横切分析与重构工具 ArchiRefactoring, 主要包括 4 个模块: 横切分析、AO-ADL 体系结构描述语言解析、构件行为(CSP 描述)解析与体系结构重构。ArchiRefactoring 工具根据输入的特征-构件追踪关系矩阵来识别横切特征, 然后根据通知类型(adviceType)将横切构件从初始体系结构中提取出来, 从而实现了体系结构的重构。

4.2 实验

针对 TMS 系统, 根据第 3 节横切特征分析, 特征模型中特征 Log 与 Authentication 是横切特征。从表 2 可知, 它们对应的直接追踪关系构件是 Log 与 Authentication, 根据我们的方法, 初始体系这两个构件需要以方面化提取出来。显然, Log 横切类型是 After, Authentication 横切类型是 Around。采用 3.3 节的重构方法, 重构后的体系结构见图 7, 其中深色框为方面构件。对比重构前后体系结构, 主要有以下变化:

- 1) 描述发生变化的构件: Receiver UI, Scheduler UI, Create Order Process 与 Schedule Order Process。
- 2) 描述发生变化的连接器: 1, 2, 10 与 14。
- 3) 删掉的连接器: 5, 6, 9 与 15。

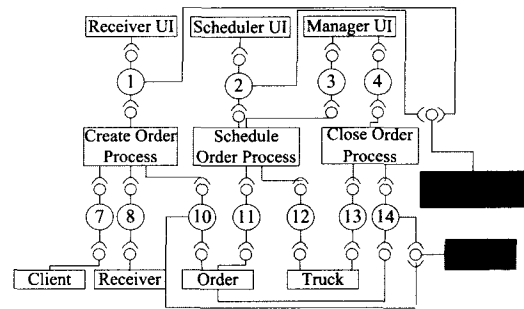


图 7 重构后体系结构

我们给出了重构前后行为变化的构件行为对比, 见表 4。限于篇幅, 只给出了构件 Create Order Process 与连接器 10 的重构前后描述, 其它的类似。Create Order Process 重构前后描述见图 8, 连接器 10 重构前后描述见图 9。图 8 中, 可以看到构件 Create Order Process 重构前后行为的变化, 同时去除了调用 Log 的接口; 图 9 中带有下划线部分是重构后新加的, 新加的部分描述了创建任务后记录 Log 的横切行为。

表 4 重构前后构件行为对比

构件	行为	重构前	重构后
Receiver UI	Behaviour=CreateTAR, checkAuthentication->>(OrderCreateUI, creatOrderII, CreateError, createOrderErrorHandling) ->> Behaviour□/√	Behaviour = OrderCreateUI, creatOrder ->> Behaviour□/√	Behaviour = OrderCreateUI, creatOrder ->> Behaviour□/√
Scheduler UI	Behaviour=CloseTAR, checkAuthentication->>(OrderDispUI, dispatchOrderII, DispError, dispatchOrderErrorHandling) ->> Behaviour□/√	Behaviour = OrderDispUI, dispatchOrder ->> Behaviour□/√	Behaviour = OrderDispUI, dispatchOrder ->> Behaviour□/√
Create Order Process	Behaviour=ClientRole, setClient->>Receiver, setReceiver->>OrderCreateManageRole, Log ->> Behaviour□/√	Behaviour = ClientRole, setClient ->> Receiver, setReceiver ->> OrderCreateManageRole, createOrder ->> Behaviour□/√	Behaviour = ClientRole, setClient ->> Receiver, setReceiver ->> OrderCreateManageRole, createOrder ->> Behaviour□/√
Close Order Process	Behaviour=ResouseRole, realseTruck ->>OrderRole, closeOrder->> OrderCloseLog log ->> Behaviour□/√	Behaviour = ResouseRole, realseTruck ->> OrderRole, closeOrder ->> OrderCloseLog log ->> Behaviour□/√	Behaviour = ResouseRole, realseTruck ->> OrderRole, closeOrder ->> OrderCloseLog log ->> Behaviour□/√

```

<interface name="ClientManage"><operation name="setClient"/>
</interface>
    
```

4 工具实现及实验

4.1 工具实现

```

<interface name = "ReceiverManage"><operation name = "setReceiver"/></interface>
<interface name = "OrderCreate"><operation name = "createOrder"/></interface>
<interface name = "CreateOrderLog"><operation name = "log"/></interface>
<interface name = "CreateOrderUI"><operation name = "creatOrderBusiness"/></interface>
(! --重构前描述-- )
<component name = "Create Order Process">
<required-interface role = "ClientRole" uri = "//interface{@name = "ClientManage"}/>
<required-interface role = "ReceiverRole" uri = "//interface{@name = "ReceiverManage"}/>
<required-interface role = "OrderCreateManageRole" uri = "//interface{@name = "OrderCreate"}/>
<required-interface role = "LogCreateOrderRole" uri = "//interface{@name = "CreateOrderLog"}/>
<provided-interface role = "OrderCreateBPRole" uri = "//interface{@name = "CreateOrderUI"}/>
<behaviour>Behaviour = ClientRole. setClient -> ReceiverRole. setReceiver -> OrderCreateManageRole. createOrder -> LogCreateOrderRole. log -> Behaviour □√
</behaviour>
</component>
(! --重构后描述-- )
<component name = "Create Order Process">
<required-interface role = "ClientRole" uri = "//interface{@name = "ClientManage"}/>
<required-interface role = "ReceiverRole" uri = "//interface{@name = "ReceiverManage"}/>
<required-interface role = "OrderCreateManageRole" uri = "//interface{@name = "OrderCreate"}/>
<provided-interface role = "OrderCreateBPRole" uri = "//interface{@name = "CreateOrderUI"}/>
<behaviour>Behaviour = ClientRole. setClient -> ReceiverRole. setReceiver -> OrderCreateManageRole. createOrder -> -> Behaviour □√
</behaviour>
</component>

```

图8 构件 Create Order Process 重构前后描述

```

<connector name = "c10">
<provided-role name = "OrderCreateProvided">
<role-specification>/component/required-interface[@role = "OrderCreateManageRole"]</role-specification>
</provided-role>
<required-role name = "OrderCreateRequired">
<role-specification>/component/provided-interface[@role = "CreateOrderRole"]</role-specification>
</required-role>
<aspectual-role name = "CreateOrderLogRole">
<role-specification>/component/provided-interface[@role = "CreateLogRole"]</role-specification>
</aspectual-role>
<componentBindings>
<binding name = "c10CreateOrderBinding">

```

```

/connector[@name = "c10"]/provided-role[@name = OrderCreateProvided] and/connector[@name = "c9"]/required-role[@name = OrderCreateRequired]
</binding>
</componentBindings>
<aspectBindings>
<aspectual-binding name = "c10 LogBinding">
<pointcut-specification>
/connector[@name = "c10"]/binding[@name = "c10CreateOrderBinding"] and/operation[@name = "createOrder"]
</pointcut-specification>
<binding operator = "after">
<aspectual-component
aspectual-role-name = "/connector[@name = "c10"]/aspectual-role[@name = "CreateOrderLogRole"]"
advice-label = "/operation[@name = "log"]"/>
</binding>
</aspectual-binding>
</aspectBindings>
</connector>

```

图9 连接器 10 重构前后描述

从这些对比可以看出,我们将两个横切特征对应的直接追踪关系构件 Log 与 Authentication 在初始体系结构上抽取了出来,表示为方面构件,并且横切关系也很明显,主要表现在连接器 1,2,10 与 14 上。这有效减少了体系结构的复杂性,很好地体现了体系结构上的横切行为,达到了重构目的。

5 讨论

5.1 特征-构件追踪关系问题

追踪关系矩阵用来表示软件开发不同时期制品之间的追踪关系,我们对其进行了扩展,引入了直接追踪关系的概念。这样,在横切特征分析后,就可以指导体系结构重构时哪些构件是横切的,需要以方面提取出来。但是,这种扩展是基于以下假设:第一,特定特征存在有直接追踪关系的构件;第二,与特定特征有直接追踪关系的构件只有一个,若有多个,可以将它们看成一个复合构件。

5.2 构件行为协议描述问题

AO-ADL 语言提供了扩展机制。AO-ADL 中构件有自己的行为描述,但是没有指定行为描述方法。我们采用 Wright 体系结构描述语言描述构件的行为,然后开发一个行为协议解析器,用于解析构件行为,得到的行为解析结果保存在自动机中,以便体系结构的自动重构。

5.3 连接器中方面绑定类型问题

AO-ADL 语言中的横切关系体现在连接器的描述上,包括方面角色与方面绑定。方面角色必须对应构件的服务接口,因为服务接口中的方法就是通知。当然,该构件可以有请求接口。另外,方面构件绑定中有切入点定义、通知类型与通知,这与 AOP 是类似的。我们的重构方法中识别 Before, After 与 Around 3 种类型,与 AspectJ 是相对应的。

结束语 软件体系结构中存在除计算与通信之外的其它类型横切关注点,这些横切行为的存在不但增加了体系结构

(下转第 183 页)

Fedora Core 5 的 IBM ThinkPad T42 笔记本上进行。使用的测试数据是通过 Galicia(<http://www.iro.umontreal.ca/~galicia/>)随机生成的两组数据。第 1 组由对象数目×属性数目分别是 $50 \times 20, 50 \times 25, 50 \times 30, \dots, 50 \times 200$ 的 37 组不同的形式背景组成。形式背景填充率($|R|$ 除以 $|O| \times |A|$)都为 30%。我们让对象的数目不变,属性数目不断增加。第 1 组数据的算法总时间和为每个节点生成的所有子节点函数所花时间分别如图 4、图 5 所示。第 2 组由对象数目×属性数目分别是 $50 \times 20, 100 \times 20, 150 \times 20, \dots, 3000 \times 20$ 的 50 组不同的形式背景组成。形式背景填充率都为 30%。这组数据我们让属性的数目不变,对象数目不断增加。第 2 组数据的算法总时间和为每个节点生成的所有子节点函数所花时间分别如图 6、图 7 所示。

比较这几幅图,可以发现当属性的个数不变时,随着概念个数的增加,CMCG 算法比 Lattice 算法越来越快。从图 8 中也可以发现,用 Trie 树来代替搜索树,也能一定程度上提高算法的时间性能。

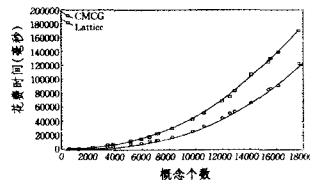


图 8 第 2 组数据 CMCG 算法和 Lattice 算法相应地在有向图 G 中插入顶点和插入边的函数所花时间

结束语 目前,构造概念格的算法很多,本文引入了概念矩阵,基于该矩阵,利用秩把这个概念的所有后继结点都找出来。同时用实验证明了 CMCG 算法比 Lattice 算法要快。实验中还发现搜索一个概念是否已经在图中存在的操作也花费了大量的时间,如何减少这个搜索时间也是值得进一步去研究的。

(上接第 146 页)

的复杂性,而且不利于维护与演化,软件产品线中尤其如此。我们提出的基于横切特征的体系结构重构方法,分为横切特征分析与体系结构重构两个阶段。本方法能在软件需求阶段发现横切特征,然后指导初始体系结构横切方面的提取,实现体系结构方面化重构。在本方法的基础上,我们实现了重构工具,并对运输管理系统进行了实验。实验结果表明我们的方法是可行的,能够自动进行横切分析与体系结构的重构。

我们的重构方法在体系结构层没有涉及到代码实现层,以后的工作就是在代码层研究重构。同时,重构前后的量化比较指标也是重要的。

参考文献

- [1] Zhang Wei, Mei Hong, Zhao Haiyan. A Feature-oriented Approach to Modeling Requirements Dependencies[C]//Proc. of the 13th IEEE International Conference on Requirements Engineering. USA: IEEE, 2005: 1-3
- [2] Shaw M, Garlan D. Software Architectures: Perspectives on an Emerging Discipline[M]. USA: Prentice Hall, 1996
- [3] Pinto M, Fuentes L. AO-ADL: An ADL for Describing Aspect-oriented Architectures [C] // Early Aspects 2007 Workshop,

参考文献

- [1] Wille R. Restructuring lattice theory: An approach based on hierarchies of concepts[C]//Rival I. Ordered Sets. Dordrecht, Boston; Reidel, 1982: 445-470
- [2] Oosthuizen G D. The application of concept lattice to machine learning[R]. South Africa: University of Pretoria, 1996
- [3] Godin R. Incremental concept formation algorithm based on Galois (concept) lattices [J]. Computational Intelligence, 1995, 11 (2): 246-267
- [4] Carpineto C, Romano G. Galois: an order-theoretic approach to conceptual clustering [C] // Proceedings of ICML-93. Utgoff Ped, 1993: 33-40
- [5] Ho T B. Incremental conceptual clustering in the framework of Galois lattice[C]// LU Hong-Jun, Motoda H, LIU Huan, eds. KDD: Techniques and Applications. Singapore: World Scientific, 1997: 49-64
- [6] Ganter B. Two basic algorithms in concept analysis. FB4-Preprint No. 831, TH Darmstadt
- [7] Stumme G, Taouil R, Bastide Y, et al. Fast computation of concept lattices using data mining techniques[C]// Proceedings of 7th International Workshop on Knowledge Representation Meets Databases (KRDB 2000). Berlin, 2000: 129-139
- [8] Nourine L, Raynaud O. A fast algorithm for building lattices[C]// Workshop on Computational Graph Theory and Combinatorics. Victoria, 1999: 199-204
- [9] Bordat J P. Calcul pratique du treillis de Galois d'une correspondance[J]. Math. Sci. Hum., 1996: 31-47
- [10] Lindig C. Fast concept analysis [C]// Stumme G. ed, Working with Conceptual Structures-Contributions to ICCS 2000. Shaker Verlag, Aachen, Germany, 2000
- [11] Ganter B, Wille R. Formal Concept Analysis: Mathematical Foundations[M]. Berlin: Springer, 1999
- [12] 翟岩慧, 曲开社, 曹桃云. 基于矩阵秩的概念格生成算法[J]. 电脑开发与应用, 2006, 19(5): 11-12

LNCS 4765. 2007: 94-114

- [4] van den Berg K, Conejero J M, Hernandez J. Analysis of Cross-cutting in Early Software Development Phases Based on Traceability[C]//Transactions on AOSDIII, LNCS 4620. 2007: 73-104
- [5] Clarke S, Baniassad E. Theme: An Approach for Aspect-oriented Analysis and Design[C]//Proc. of the 26th International Conference on Software Engineering. 2004
- [6] Baniassad E, Moreira A, Araujo J, et al. Discovering early aspects[J]. IEEE Software, 2006, 23(1): 61-70
- [7] Allen R. A Formal Approach to Software Architecture [D]. USA: School of Computer Science Carnegie Mellon University, 1997
- [8] Allen R, Garlan D. A Formal Basis for Architectural Connection [J]. ACM Transactions on Software Engineering and Methodology, 1997, 6(3): 213-24
- [9] Peng Xin, Zhao Wenyun, Xun Yunjiao, et al. Ontology-based Feature Modeling and Application-oriented Tailoring [C] // ICSR LNCS 4039. 2006: 87-100
- [10] 张伟, 梅宏. 一种面向特征的领域模型及其建模过程[J]. 软件学报, 2003, 14(8)
- [11] 孙昌爱, 金茂忠, 刘超. 软件体系结构研究综述[J]. 软件学报, 2002, 13(7)