# 从面向方面程序设计的定义到面向方面程序设计语言

古思山[1] 蔡树彬[2] 李师贤[1]
(中山大学信息科学与技术学院 广州 510275)[1] (深圳大学计算机与软件学院 深圳 518060)[2]

**摘 要** 目前无论在学术界还是工业界,很多人士简单地将面向方面程序设计理解为模块化横切关注点,狭隘地认为面向方面程序设计构建在面向对象程序设计之上,是面向对象程序设计的扩展、有效的补充等。回顾了面向方面程序设计的定义,分析了面向方面程序设计区别于其它程序设计方法的本质特征,并形式化了定义中的量化语句与方面,给出了量化语句与方面的语义。重申了面向方面程序设计的本质特性是Quantification与Obliviousness,模块化横切关注点只是其带来的一个好处。同时指出了面向方面程序设计是一种新型的程序设计范式,独立于面向对象等其它程序设计方法。并且,基于此定义,给出了面向方面程序设计语言需满足的最小条件集,并比较了主流的面向方面程序设计语言,探讨了这些语言在面向方面程序设计理念上的差异。

**关键词** 面向方面程序设计,面向方面程序设计语言,横切关注点,Quantification,Obliviousness
**中图法分类号** TP311 **文献标识码** A

## From the Definition of Aspect-oriented Programming to Aspect-oriented Programming Languages

GU Si-shan[1] CAI Shu-bin[2] LI Shi-xian[1]
(School of Information Science and Technology, Sun Yat-sen University, Guangzhou 510275, China)[1]
(School of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China)[2]

**Abstract** Today a lot of people not only from industrial community but also from academic community simply take Aspect-Oriented Programming (AOP) as modularizing crosscutting concerns and in a narrow-mined way to believe that AOP is just an extension or an effective supplement to Object-Oriented Programming (OOP). Based on the definition of AOP, its nature which makes it different from the other programming languages was dug out. And the quantified statement and aspect in the definition were formalized. Moreover the semantics of them were defined. And then we argued quantification and obliviousness in the definition are the real nature of AOP. Modularizing crosscutting concerns is just a benefit from it. And AOP is a new programming paradigm which is independent of all the other programming languages. Based on the definition, the minimum condition set which Aspect-Oriented Programming Languages (AOPL) need to satisfy was proposed. And the difference between the mainstream AOPL from the view of the definition was probed into.

**Keywords** Aspect-oriented programming, Aspect-oriented programming language, Crosscutting concerns, Quantification, Obliviousness

## 1 引言

程序设计本质上是一个问题求解过程。针对给定的问题,通过一系列的方法、手段,如分析、设计、编码、测试等,制造出可以解决该问题的软件制品。然而现实的问题往往过于复杂,直接求解会比较困难。D. L. Parnas在1972年开创性地提出了模块化思想[1],即按照一定的方法,将一个复杂的大问题分解为多个较小的问题,再将这些较小的问题分解为更小的问题。如此循环,直至每一个小问题都可以编写代码来求解;求解完每一个小问题后,再按照一定的方法将这些小问题的解决方案复合、组装成原来大问题的解决方案。这种"分而治之"的思想广为人们接受。当时提出的大问题分解为小问题的模块化思想也演化为今天我们所熟悉的关注点分离[2]。软件开发的关键体现在不断地分离关注点及不断地组合这些关注点的解决方案上。这几十年来,程序设计语言也在沿着这个方向发展,为关注点的分离及其解决方案的组合提供更加良好的机制,如结构化程序设计的子程序、面向对象程序设计的数据封装技术等等。

Parnas将模块定义为包含自己的数据和相关操作的软件实体,而且这些数据只能通过该模块操作来访问,其它的模块不能直接访问[1]。在这个理念下,现实系统的功能常常分布于多个模块,模块常常需要访问其它模块的数据或操作。

到稿日期:2010-12-19 返修日期:2011-04-23 本文受广东省自然科学基金项目(10351806001000000),深港创新圈项目(ZYB200907060012A),广东高校优秀青年创新人才培养计划项目资助(LYM09121),深圳市科技计划项目(JC200903120046A,JC201005280434A)资助。
古思山 男,博士生,主要研究方向为形式语义学、AOP、MDA等,E-mail:sishangu@gmail.com;蔡树彬 男,博士,主要研究方向为软件工程、本体等;李师贤 男,教授,博士生导师,主要研究方向为形式语义学、软件开发方法学、软件成本估算等。

而访问模块的数据需要调用该模块的操作(接口)来完成,这样就不可避免地存在大量对模块的操作(接口)的调用。Parnas 的模块化思想被直接用于面向对象程序设计。面向对象将一组相关的数据和操作封装为类(对象),类(对象)的数据只能通过其操作来访问,其它的类(对象)不能直接访问(有些面向对象程序设计语言要求没有这么严格,有些类(对象)的数据也可以被其它的类(对象)直接访问)。这也使得在面向对象的程序中,存在大量对类(对象)的操作的调用。结果造成了一方面对一些关注点(如日志、身份验证等)的操作的调用散布于整个系统中(code scattering,代码散乱);另一方面,这些散布于整个系统中的关注点又常常与其它关注点耦合在一起(code tangling,代码纠缠)。这些散布于整个系统中的关注点具有"贯穿特性"[3],也被称为横切关注点。

面向方面程序设计(Aspect-Oriented Programming,AOP)[4]提供了模块化横切关注点的机制,很好地解决了面向对象程序设计固有的代码散乱和代码纠缠问题。在面向方面程序设计中,业务逻辑被分为核心关注点(code concerns)和横切关注点(crosscutting concerns)两部分。前者涉及系统的核心业务逻辑,如 ATM 系统中的取款、转账及查询等功能;后者主要实现系统中相对独立而又非核心业务的功能,而且这些功能往往散布于整个系统,如日志记录、身份验证、事务管理等。在编写程序时,不再需要显式地调用横切关注点的操作,只需使用面向方向语言提供的机制指定需要加入横切关注点逻辑的地方(AOP 中又称为连接点),系统在执行时会自动在连接点执行横切关注点的逻辑。

自面向对象方法成熟后,对程序设计方法的研究进入较沉寂的阶段,鲜有跨跃性的成果。而面向方面程序设计很好地解决了面向对象程序设计不可避免又备受诟病的代码散乱和代码纠缠问题。无论在学术界还是在工业界都如一颗春雷,引发了巨大震动。不少学者断定面向方面程序设计将会是在程序设计方法史上继面向对象程序设计之后的又一重大变革。AOP 也就成了这些年程序设计语言领域研究的热点话题。大量面向方面程序设计语言也不断进入大众的视野。

这几年来,关于面向方面程序设计的书籍、文献不断涌现,网络上更是有大量关于面向方面程序设计的信息。然而,这些书籍、文献及其它相关信息的质量参差不齐,而且绝大多数都过于片面强调面向方面程序设计带来的模块化横切关注点,解决了面向对象程序设计固有的代码散乱和代码纠缠问题,这一好处使得如今谈及面向方面程序设计就会谈到模块化横切关注点,其几乎成了面向方面程序设计的代名词。更为遗憾的是无论在工业界还是学术界,很多人士狭隘地将面向方面程序设计理解为面向对象程序设计的延续、扩展或有效的补充,不但将面向方面程序设计绑定在面向对象程序设计上,还将面向方面程序设计局限于面向对象程序设计这个范畴。

然而模块化横切关注点只是面向方面程序设计带来的好处,属于方法学的范畴,并不是面向方面程序设计的本质特性,也就不能回答诸如"什么是面向方面程序设计?"这类问题,也就是说能否模块化横切关注点并不是判断某种语言是否属于面向方面程序设计语言的标准。可见,对于面向方面程序设计,我们太过纠结其带来的好处,而忘却了其本质的特征。也因此对其产生了诸多狭隘的见解、误解,甚至错解。

理解事物的本质才是理解事物的关键,也只有理解了事物的本质才算是真正理解了该事物。本文回顾了面向方面程序设计的定义,并形式化了此定义,给出了定义中的量化语句和方面的语义,清晰地解读了面向方面程序设计区别于面向对象程序设计等其它程序设计方法的本质特征,纠正了诸多对面向方面程序设计狭隘或错误的认识,同时还基于面向方面程序设计的定义,给出了面向方面程序设计语言需要满足的最小条件集,并且分析、比较了当今主流的面向方面程序设计语言在面向方面程序设计的理念这一层面的差异。

## 2 面向方面程序设计的定义

最为大众接受的面向方面程序设计的定义莫过于 Filman 与 Friedman 在 2000 年的 OOPSLA 关于分离关注点的讨论会上所提出的面向方面程序设计的两大特性[5],Filman 在文献[6]进一步解释并强调了这两大特性:

Aspect Oriented=quantification+obliviousness (1)

其中,Quantification 是指编程人员可以用量化的语句去定义程序的行为。具体来说,编程人员只需编写独立、一致的语句,就可以作用在基础程序的非局部的多处地方。如"每次调用方法 void fresh()之前,记录日志"。而 Obliviousness 则是指基础程序并不需要为这些额外的行为做任何准备。基础程序并不知道也不需要知道这些额外行为的存在,也就不知道这些额外的行为在什么时候执行,更不可能知道这些额外的行为的具体内容或效果。

此外,为更加详细地阐述面向方面程序设计的内在特性,Filman 与 Friedman 又做了如下说明:

AOP is thus the desire to make programming statements of the form: In programs P, whenever condition C arises, perform action A. Over 'conventionally' coded programs P (2)

其中,后面的"Over 'conventionally' coded programs P"强调程序 P 编写完成后才用量化语句修改程序的行为,这其实是为了强调 Obliviousness 特性。程序 P 并不知道后来才编写的量化语句。从程序 P 的代码中,看不出是否有方面执行、什么时候执行等信息。理论上,也可以先编写好量化语句,然后再编写基础程序 P。这时要求基础程序 P 严格按照足够详细的编程规范和要求来编写,以保证量化语句指定的行为在正确的地方或正确的时候执行。中间那句则形象地说明了 Quantification 特性。在 AOP 中,这些后加的指定程序行为的量化语句称为方面。

AOP 的核心思想是去除传统程序设计方法(结构化程序设计、面向对象程序设计)中需要其它模块来参与完成某些功能时,必须在程序中显式地给出调用语句这一必要条件,允许编程人员在已编写好的程序上指定在某些条件发生时执行某些行为。

## 3 形式化方面

根据式(2),可设 $C$ 表示条件的全集,$A$ 表示行为的全集,量化语句定义为二元组:

$$\langle C,A\rangle \in C\times A$$

式中,$C$ 为量化语句的条件,$A$ 为量化语句的行为。设 $S$ 表示量化语句的全集,定义量化语句的条件函数 $con: S\to C$ 及行为函数 $act: S\to A$,且:

$con(\langle C,A\rangle)=C \quad act(\langle C,A\rangle)=A$

设 $I_p$ 表示程序 P 的连接点的全集。定义条件到程序 P 的连接点的映射函数 $jpC:C\rightarrow P(I_p)$，返回条件指定的全部连接点，其中 $P(I_p)$表示 $I_p$ 的幂集，下同。

定义量化语句到程序 P 的连接点的映射函数 $jpS:S\rightarrow P(I_p)$

$jpS(\langle C,A\rangle)=jpC(con(\langle C,A\rangle))$

连接点及需要在该点执行的行为是量化语句最终描述的内容，可理解为其语义。可将量化语句的语义域定义为 $P(I\times A)$，并定义语义函数 $\mu:S\rightarrow P(I\times A)$：

$\mu(\langle C,A\rangle)=\{\langle I,A\rangle \mid I\in jpS(\langle C,A\rangle)\}$

一个方面可以有多条独立的量化语句。方面可看成是由量化语句组成的集合(此时并没有考虑量化语句的顺序)。设 $AS$ 表示方面的全集。扩展语义函数 $\mu$ 至方面，即 $\mu:S\cup AS\rightarrow P(I\times A)$。设方面 $AS\in AS$，其语义定义为：

$\mu(AS)=\{\langle I,A\rangle \mid S\in AS\wedge\langle I,A\rangle\in\mu(S)\}$

方面的语义是方面包含的量化语句的语义的并集。

AOP 的 Quantification 特性允许编程人员编写一条或多条独立的量化语句，以便在基础程序非局部的多处发生作用。而传统的结构化方法中的函数调用或面向对象方法中的方法调用在每次调用时都需要编写一条调用语句。可见，AOP 的 Quantification 特性避免了执行相同行为的连接点的一一声明。AOP 的 Obliviousness 特性则允许基础程序不需要为后来附加的量化语句的行为做任何准备。这就去除了传统结构化方法中的函数调用及面向对象方法中的方法调用时每次都需要显式地给出调用语句这个必要条件。

AOP 之所以能模块化横切关注点，首先应归功于其 Obliviousness 特性，使得基础程序去除了原来散布在系统各处对横切关注点的调用，把横切关注点对基础程序的作用封装在方面中。这就解决了代码散乱和代码纠缠的问题。其次，AOP 的 Quantification 特性支持编程人员使用简洁的语言来指定横切关注点的切入，避免了连接点的一一罗列，从而简化了方面的开发。

## 4 从面向方面程序设计的定义看面向方面程序设计语言

### 4.1 面向方面程序设计语言的需求

从面向方面程序设计的定义，即式(1)及式(2)，可以得出面向方面程序设计语言必须满足以下条件：

1)描述基础程序 P 的语言：程序 P 是面向方面程序的基础程序，面向方面程序设计的量化条件 C 和行为 A 都针对基础程序 P。很多面向方面程序设计语言直接采用现有的程序设计语言作为描述基础程序 P 的语言，如 Java、C++、C#、C 等。

2)描述量化条件 $C$ 的语言：面向方面程序设计采用量化条件来指定行为 $A$ 在基础程序 P 中的诸多连接点。量化条件 $C$ 的描述语言决定了可以量化的基础程序语言的元素、这些元素的组合方式以及连接点的上下文等，从而影响到面向方面程序设计语言的描述能力、易用程度等性质。

3)行为 $A$ 的描述语言：行为 $A$ 是程序设计人员期望在基础程序 P 的连接点处执行的行为。可以说是在基础程序 P 上引入方面的最终目的。理论上，行为 $A$ 的描述语言和基础程序 P 的描述语言没有直接的关系，可以相同，也可以完全不同。事实上，考虑到实现的难易程度及与基础程序 P 的交互等因素，绝大多数面向方面程序设计语言都是直接采用基础程序 P 的描述语言作为行为 A 的描述语言。

4)方面功能的实现：面向方面程序设计语言需要实现方面的功能。程序执行时，在量化语句指定的连接点按要求执行行为 $A$。这在 AOP 中称为编织(weaving)。有些面向方面程序设计语言采用预处理等机制修改基础程序 P 的源代码或中间代码(如 Java 的字节码)来实现；也有些面向方面程序设计语言采用动态代理、运行时事件监控等方式来实现。

这些条件是面向方面程序设计语言的最小条件集。如果某个语言不满足这个条件集的任一条件，那么这个语言就不是面向方面程序设计语言。当然，除了满足这个最小条件集外，面向方面程序设计语言还可以根据需要提供其它方面的能力。如 AspectJ[7] 的 Inter-type Declarations 机制可以修改类的静态结构，包括增加类的属性、方法及增加类的父类、实现的接口等。

### 4.2 基础程序语言

式(1)说明了 Quantification 和 Obliviousness 是面向方面程序设计内在的两个特性，而式(2)形象地描述了这两个特性。无论是式(1)还是式(2)，都没有对基础程序 P 的语言做任何的限制，也就是说面向方面程序设计与其基础语言没有直接的关系。从这个意义上来说，基础程序 P 的语言可以是任何编程语言。也就是说面向方面程序设计可以构建在任何编程语言上，而不仅仅局限于面向对象程序设计语言。可见，广为流传的 AOP 构建在 OOP 之上，AOP 是 OOP 的扩充、AOP 是 OOP 的延续、AOP 是 OOP 有效的补充等观点是对 AOP 局限性的理解。这些观点之所以能广为流传，被大众所接受，主要原因有 2 个。首先，AOP 最初的出发点就是解决 OOP 固有的代码散乱和代码纠缠问题。这很容易给人 AOP 是 OOP 的扩充、AOP 是 OOP 的延续等错觉；其次，早期出现的面向方面程序设计语言如 AspectJ、Spring AOP[9]、JBoss AOP[10] 等都是以面向对象程序设计语言为基础的，而且最为流行的几种面向方面程序设计语言也都基于面向对象程序设计语言，这使得人们不自觉地会把面向方面程序设计和面向对象程序设计搅在一起。而事实上，面向方面程序设计是一种独立于其它程序设计方法的新型的程序设计方法。

表 1 面向方面程序设计语言

| 基础语言 | 面向方面语言 |
|---|---|
| Java | AspectJ, AspectWerkZ, Spring AOP, JBoss AOP, JAC, CaesarJ, Compose*, Dynaop |
| .NET Framework | LOOM. NET, AspectDNG, Aspect #, Spring. NET, Compose*, PostSharp, Seasar. NET |
| C 或 C++ | AspectC++, FeatureC++, AspectC, AspeCt-oriented C, Aspicere |
| Common Lisp | AspectL |
| Deiphi | InfraAspect, MeAOP in MeSDK |
| JavaScript | Advisable, Ajaxpect, JQuery AOPPlugin, AspectJS, Joose, Aspectes |
| ML | AspectML |
| PHP | PHPaspect, Aspect-Oriented PHP, PHP-AOP |
| Python | PEAK, Aspyct AOP, Lightweight Python AOP, Pythius |
| Ruby | AspectR, AspectR-Fork, Aquarium |
| Squeak Smalltalk | AspectS, Metaclass Talk |

从表1可以看出，面向方面程序设计与基础语言没有直接的关系。面向方面程序设计语言除了可以构建在面向对象程序设计语言之上，如Java、C++、.Net Framework中的C#、VB.NET等；还可以构建在结构化程序设计语言之上，如C；也可以构建在函数式程序设计语言之上，如ML；甚至可以构建在脚本语言、动态语言之上，如JavaScript、PHP、Python、Ruby等。

另一方面，无论是工业界还是学术界，对基于面向对象程序设计语言的面向方面程序设计语言关注较高，研究和应用都较多，相关的支撑工具也较丰富。而其它的面向方面程序设计语言大多数都是研究团体或个人的研究成果，还处于启蒙、初始阶段。有些语言连文档都还很不完善，更不用说有关软件开发的支撑工具。从语言的成熟度及其应用性来看，基于面向对象程序设计语言的面向方面程序设计语言显得一枝独秀。下面在讨论面向方面程序设计语言时也主要针对比较成熟的面向方面程序设计语言，其中大部分都基于面向对象程序设计语言，如AspectJ、AspectWerkz[8]、Spring AOP、JBoss AOP与JAC[11]基于Java；ApectC++[12]基于C++；Aspect.NET[13]基于.NET平台；AspcetC[14]基于C语言。

### 4.3 条件C可量化的元素

式(1)和式(2)都没有对条件$C$的量化对象做任何的限制。条件$C$是对基础程序P的量化，描述了行为$A$在基础程序P中的连接点。理论上，基础程序P中可以作为连接点的元素都可以是量化的对象。事实上，不同的基础语言会有不同的可供量化的元素，如面向对象程序设计语言中的方法调用、对象属性访问、对象创建与销毁、类型匹配、异常抛出等，结构化程序设计语言中的函数调用以及两者都有的变量定义、变量访问、变量比较等细粒度的元素。表2列出条件$C$可量化的元素。

表2 条件$C$可量化的元素

| 量化元素 \ AOP语言 | | Aspect | AspectJ WerkZ | Spring AOP | JBoss AOP | JAC | Aspect C++ | Aspect Net | AspectC |
|---|---|---|---|---|---|---|---|---|---|
| 静态 | 方法调用 | √ | √ | √ | √ | √ | √ | √ | √ |
| | 属性访问 | √ | √ | | √ | √ | | √ | |
| | 对象创建 | √ | √ | | √ | √ | √ | √ | |
| | 对象销毁 | | | | | | √ | | |
| | 类型匹配 | √ | √ | | √ | √ | √ | √ | |
| | 变量定义 | | | | | | | | |
| | 变量访问 | | | | | | | | |
| | 变量比较 | | | | | | | | |
| | 抽象语法树 | | | | | | | | |
| | 异常抛出 | √ | √ | √ | √ | √ | | √ | |
| 动态 | 堆栈大小 | | | | | | | | |
| | 使用轨迹 | % | % | % | % | % | % | | % |

标记说明：√表示支持，%表示部分支持，空白表示不支持，下同。

除了AspectC是基于结构化程序设计语言C外，其它的都是基于面向对象程序设计语言。AspectC只提供了结构化程序设计中最常用的元素——函数作为连接点，绝大多数基于面向对象程序设计语言的面向方面程序设计语言都支持将方法调用、属性访问、对象创建、类型匹配及异常抛出作为连接点。对象销毁在Java中通过自动垃圾回收机制实现，编程人员无需关注，这也使得基于Java的面向方面程序设计语言都不支持将对象销毁作为连接点。C++没有异常处理机制，使得基于C++的面向方面程序设计语言也就不支持将异常抛出作为连接点。

几乎所有的面向方面程序设计语言都不支持将更细粒度的变量定义(C与C++还区分全局变量与局部变量)、变量访问、变量比较等作为连接点，原因在于面向方面程序设计源于面向对象程序设计。而在面向对象程序设计中，类是基本组成单元，封装了属性及方法。系统的状态由运行时对象的状态决定。而来自结构化程序设计中的变量这一概念已经淡化，只存在于方法体之中。这些变量只是方法执行时起辅助作用的临时变量，并不影响系统的状态。这就使得将变量相关的操作作为连接点意义不大。这种思想也影响了后续出现的面向方面程序设计语言。

同样，几乎所有的面向方面程序设计语言都不支持将基于抽象语法树的条件作为连接点。原因在于抽象语法树是编译时的概念，程序设计人员及实现人员都无需了解。量化语句属于设计或实现这一层次，也就无需了解编译相关的抽象语法树。

面向方面程序设计语言对动态时的元素作为连接点支持最多的是异常抛出。而对调用轨迹的支持最初源自AspectJ的cflow机制，只是稍稍涉及到调用轨迹。与调用轨迹相关的稍为复杂的条件，如调用方法$A$后再调用方法$B$、调用方法$A$后最终会调用方法$B$等，都不支持。

### 4.4 条件C的描述形式

式(1)和式(2)中都没有对条件$C$的描述形式做任何的限制。描述形式在使用上的难易程度直接影响到面向方面程序设计语言的易用性，而在实现上的难易程度也会影响到面向方面程序设计语言实现的难易程度。当然，描述形式的难易程度与其描述能力也有关系。一般，描述能力越强，其描述形式也会越复杂。表3列出条件$C$的描述形式。

表3 条件$C$的描述形式

| 描述形式 \ AOP语言 | AspectJ | Aspect WerkZ | Spring AOP | JBoss AOP | JAC | Aspect C++ | Aspect Net | AspectC |
|---|---|---|---|---|---|---|---|---|
| XML | √ | √ | √ | √ | | | √ | |
| Annotation | √ | √ | | √ | | | √ | |
| 自定义语言 | √ | | | | √ | √ | | √ |

条件$C$的描述形式可以简单分为3大类：一类是XML，另一类是使用基础语言提供的Annotation机制，最后一类则是自定义一套语言。XML是声明式语言，格式比较随便；与之相比，自定义语言往往更像是代码风格，格式要求严格；而Annotation采用的是注释的形式，有一定的格式要求，严格性位于前面两者之间。

从易用性来看，XML为大众所熟悉，3者之中易用性最高；Annotation作为基础语言的一部分，也为编程人员所熟悉，易用性次之；在使用自定义的语言之前，设计人员或编程人员都需要先熟悉这门自定义的语言，易用性最差。从实现上来说，XML的解析有标准化的方法及实现；Annotation作为基础语言的一部分，基础语言本身就提供了解析的方法，这两类描述形式实现起来都相对容易；自定义语言时需要实现相应的编译器或解析器。绝大多数采用自定义语言的实现都是在基础语言之上扩展，也相应地扩展基础语言的编译器或解析器，工作量比较大，难度也比较高。而从使用的效果来看，自定义的语言往往类似基础语言，语法简单明了，可减少

键入错误，加上可在扩展的编译器上附加代码检查等功能，很多错误可以在编译时发现，进一步减少了运行时的错误；与之相比，XML 的描述内容较杂乱，而且语法检查能力也较弱，使用时比较容易出错；Annotation 在格式要求及语法检查能力上都在前面两者之间。

### 4.5 条件 C 提供的上下文

在传统的结构化程序设计的函数调用及面向对象程序设计的方法调用中，被调用函数或方法能够访问的上下文决定于该函数或方法的参数。调用者在发起调用时需要提供被调用方需要的参数。被调用方可以修改这些参数，从而可能影响到整个程序。而调用者则能够获得由被调用者提供的返回值。在面向方面的程序中，基础程序并不知道方面的存在，也就不需要为方面的行为的执行做任何的准备，而且基础程序并不期望从方面的行为获得返回值。如果编程人员想通过方面的行为影响系统状态，就只能通过访问条件 C 所提供的上下文来实现。

式(1)和式(2)都没有对条件 C 可以提供的上下文做任何限制。条件 C 可以为行为 A 提供的上下文的能力直接影响到面向方面程序设计语言对基础程序的影响能力。上下文由量化语句指定的连接点提供。显然，不同的连接点会有不同的上下文。根据访问性质的不同，可以将上下文的访问分为读与写两种访问方式。读操作不会改变上下文，也就不会影响到系统的状态；而写操作则可能会改变上下文，从而改变系统的状态。表 4 列出条件 C 提供的上下文。

表 4 条件 C 提供的上下文

| 上下文 \ AOP 语言 | AspectJ | Aspect WerkZ | Spring AOP | JBoss AOP | JAC | Aspect C++ | Aspect Net | AspectC |
|---|---|---|---|---|---|---|---|---|
| 读 全部对象 | | | | | | | | |
| 读 调用者 | √ | √ | √ | √ | √ | √ | √ | |
| 读 被调用者 | √ | √ | √ | √ | √ | √ | √ | |
| 读 局部对象 | | | | | | | | |
| 读 局部变量 | | | | | | | | |
| 读 参数 | √ | √ | √ | √ | √ | √ | √ | √ |
| 读 堆栈大小 | | | | | | | | |
| 读 调用轨迹 | | | | | | | | |
| 写 全部对象 | | | | | | | | |
| 写 调用者 | √ | √ | √ | √ | √ | √ | √ | |
| 写 被调用者 | √ | √ | √ | √ | √ | √ | √ | |
| 写 局部对象 | | | | | | | | |
| 写 局部变量 | | | | | | | | |
| 写 参数 | √ | √ | √ | √ | √ | √ | √ | √ |
| 写 堆栈大小 | | | | | | | | |
| 写 调用轨迹 | | | | | | | | |

其中调用者、被调用者是指连接点为函数或方法调用时的调用者与被调用者；局部变量和局部对象是函数或方法调用时该函数或方法内部的局部变量和局部对象；参数的涉及面较广。连接点为函数或方法调用时参数指的是该函数或方法的参数；连接点为属性访问时参数指的是该属性；连接点为异常抛出时参数指的是捕获到的异常对象。

在对上下文环境的处理上，主流的面向方面程序设计语言表现出出奇的相似。基本上都提供了也只提供了对参数的读写操作以及在方法调用时对调用者和被调用者的读写操作。而对系统的全部对象、局部对象、局部变量、堆栈大小以及调用轨迹等都不列为可以访问的上下文。可以看出面向方面程序设计中条件 C 描述的上下文的设计受传统结构化程序设计和面向对象程序设计中函数或方法调用时的上下文的设计影响很深。

### 4.6 条件 C 描述的时间

式(2)中条件的描述并没有排除可以做时间上的约束。事实上，描述条件时也经常需要有时间上的约束，才能更加准确、完整地描述条件。如量化语句“每次调用方法 void fresh()时，记录日志”，并没有具体指明“记录日志”是发生在方法 void fresh()执行之前还是之后，还是该方法执行前后都需要执行“记录日志”这个行为。当然也可以限定一段时间后才执行方面的行为。如“每次执行完方法 void fresh()的 5 分钟后，记录日志”，表示“记录日志”的行为应该发生在执行完方法 void fresh()的 5 分钟后。表 5 列出条件 C 描述的时间。

表 5 条件 C 描述的时间

| 时间 \ AOP 语言 | AspectJ | Aspect WerkZ | Spring AOP | JBoss AOP | JAC | Aspect C++ | Aspect Net | AspectC |
|---|---|---|---|---|---|---|---|---|
| Before | √ | √ | √ | √ | √ | √ | √ | √ |
| After | √ | √ | √ | √ | √ | √ | √ | √ |
| Around | √ | √ | √ | √ | √ | √ | √ | √ |
| Interval | | | | | | | | |

在对条件 C 可以描述的行为 A 相对于连接点的执行时间的处理上，主流的面向方面程序设计语言再次表现出了出奇的相似性，它们都支持在连接点之前、之后或前后都执行行为 A，而对时间段则都不支持。一方面，受最先出现的面向方面程序设计语言 AspectJ 的影响，后续出现的面向方面程序设计语言都向 AspectJ 看齐；另一方面，时间段的实现会比较复杂，而且实际应用中对时间段的需求也比较少，而对前面 3 种时间描述的实现会相对容易，且在实际应用中也往往能够满足用户的需要。

### 4.7 方面的动态化

式(2)中只要求基础程序 P 在编写量化语句之前完成，并没有限制编写量化语句时，基础程序是否正在运行。也就是说是否支持在基础程序 P 运行过程中，编写量化语句去补充、修改程序的行为是面向方面程序设计语言可以选择的。表 6 列出方面的动态化。

表 6 方面的动态化

| 时间 \ AOP 语言 | AspectJ | Aspect WerkZ | Spring AOP | JBoss AOP | JAC | Aspect C++ | Aspect Net | AspectC |
|---|---|---|---|---|---|---|---|---|
| 增加 | | √ | | | √ | | | |
| 修改 | | √ | | | √ | | | |
| 移除 | | √ | | | √ | | | |

AspectWerkZ 和 JAC 提供了在基础程序 P 运行过程中增加、修改及移除方面的能力，对方面的动态化支持较好。而其它的几种主流的面向方面程序设计语言则没有这方面的能力，这与语言的实现机制有关系。如 AspectJ、AspectC++、AspectC 是通过扩展编译器的方式来实现的，编译之后程序的行为已经固定，不能够再修改了；AspectWerkZ 的运行时事件监控与 JAC 对反射机制的利用等对程序运行时的信息获取与处理机制能够为方面的动态化提供良好的支持。

## 5 相关研究

Filman 与 Friedman 在 2000 年的 OOPSLA 关于分离关

注点的讨论会上首次提出了面向方面程序设计的定义即式(1)与式(2)[5]，回答了什么是面向方面程序设计这一基础问题，也为判别某个语言是否属于面向方面程序设计语言提供了标准。此外，他们还分析了基于规则的系统(Rule-based Systems)、事件驱动与订阅接收(Event-based, Publish and Subscribe)、意图编程与元编程(Intentional Programming and Meta-programming)、产生式编程(Generative Programming)等是否属于面向方面程序设计这一范畴。在次年 ECOOP 关于多维度分离关注点的讨论会上[6]，Filman 受邀作报告，详细解释了他们之前的定义即式(1)与式(2)，并强调了 Quantification 和 Obliviousness 特性是面向方面程序设计区别于其它程序设计方法的本质特性。Filman 与 Friedman 的工作对本文启发较多，本文的工作构建在他们提出的面向方面程序设计的定义上。Filman 与 Friedman 的工作重点是给出面向方面程序设计的定义，并解释这个定义；而本文的重点在于分析面向方面程序设计的定义以纠正时下流行的诸多对面向方面程序设计的误解或错解，并分析、比较当下主流的面向方面程序设计语言在面向方面程序设计的理念这一层面的差异。

在绝大多数声音都认同面向方面程序设计可以提高程序的模块化与程序的可理解性时，Steimann 提出了不一样的见解[15]，他认为面向方面程序设计的成功是自相矛盾的。Steimann 注意到了尽管面向方面程序设计的方面封装了横切关注点，成为独立的模块，但是封装横切关注点的模块对基础程序模块的访问不是通过传统的模块与模块之间的交互机制(即调用基础模块提供的接口)来实现的，而是直接侵入式地访问基础模块。这种侵入式的访问又映射出封装横切关注点的单元不是传统意义上的模块。他还提到面向方面程序设计的实际开发过程中，方面对基础程序的依赖较高，而且基础程序和方面之间又没有接口这一部分，使得方面的开发与演化都不能独立于基础程序进行，这违反了软件开发方法的一个重要法则。与本文相类似的是 Steimann 也从 Filman 与 Friedman 提出的面向方面程序设计的定义出发，分析了面向方面程序设计语言可能存在的差异。不同的是 Steimann 的分析仅限于理论层次的简单探讨，没有做较深入的分析，也没有具体到实际的面向方面程序设计语言。这些都源于他的出发点在于从软件开发这个大视角来审视面向方面程序设计，而非分析或比较具体的面向方面程序设计语言。

在 2005 年的面向方面软件开发国际会议(AOSD'05)上，Kersten 分析、比较了当时主流的面向方面程序设计语言[16]。Kersten 从语法、语言机制、执行效率、语言的支撑工具、文档资料等方面分析并比较了当时主流的面向方面程序设计语言 AspectJ、AspectWerkz、JBoss AOP 以及 Spring AOP。其目的在于为实际项目的需要提供选择面向方面程序设计语言的技术参考。本文从面向方面程序设计的定义这个视点来分析并比较主流的面向方面程序设计语言，侧重于语言的设计理念这一理论层面；而 Kersten 的工作则侧重于实际应用这一层面。可以说本文的工作与 Kersten 的工作是相互很好的补充。

文献[17]综述了面向方面程序设计语言近十年来的研究进展，分析并总结了面向方面程序设计语言的主要语言特性和关键实现技术，包括连接点、切入点、通知、方面等核心概念及切入点匹配方式、Introduction 机制等，并按照不同的语言特性对面向方面程序设计语言进行了分类，如按照语法结构的对称性、逻辑的表达方式、织入行为的执行时间、织入行为的结构特征、织入行为的执行过程等。作为综述性研究，文章还探讨了面向方面程序设计语言的发展和研究方向。本文的研究与文献[17]有相交叉的地方，都对面向方面程序设计语言进行了分类。不同的是缘于不同的视角，分类的标准有所差别。文献[17]着重于语言的特性和关键技术，与 Kersten 的研究类似，关注语言的具体实现这一层面；而本文则是从面向方面程序设计的理念这一层面来分析、比较面向方面程序设计语言。本文的工作与文献[17]也是相互很好的补充。

**结束语** 面向方面程序设计这一新型的程序设计范式以其模块化横切关注点，解决了困扰面向对象程序设计已久的代码散乱与代码纠缠问题，触动了程序设计方法领域的所有人的神经，一度被认为是程序设计方法史上面向对象方法对于结构化方法的重大变迁。近几年来，面向方面程序设计领域的研究不断深入，面向方面程序设计语言越来越多，相关的支撑工具也越来越丰富，面向方面程序设计的应用也在不断扩展。这使得很多人只看到了面向方面程序设计带来的模块化横切关注点这一好处，甚至简单地将面向方面程序设计理解为模块化横切关注点；也使得很多人狭隘地将面向方面程序设计绑定、局限在面向对象程序设计之上，认为面向方面程序设计是面向对象程序设计的扩充、有效的补充等。

本文回顾了面向方面程序设计的定义，并基于此定义分析了面向方面程序设计与传统的结构化程序设计及面向对象程序设计的本质差别；指出模块化横切关注点只是面向方面程序设计带来的好处，并不是其本质特性；强调了面向方面程序设计是一种新型的程序设计范式，独立于任何程序设计方法，也可以构建在任何程序设计语言之上。

本文还形式化了面向方面程序设计定义中的量化语句与方面，给出了它们的语义，清晰地解读了面向方面程序设计的 Quantification 特性与 Obliviousness 特性；从面向方面程序设计的定义出发，给出了面向方面程序设计语言需要满足的最小条件集；分析、比较了现下主流的面向方面程序设计语言。一方面，在面向方面程序设计理念的指引下，可以更好地理解这些语言表现出来的能力、特性上的差异；另一方面，又可以通过这些具体语言表现出来的能力、特性，加深对面向方面程序设计的理念的理解。

应该注意到本文只是从面向方面程序设计的理念这个视角来分析、比较主流的面向方面程序设计语言，并没有涉及语言的实现技术、工具支持等层面。全面、系统地去探讨这些语言，分析其优缺点将是本文后续研究的重要方向。

## 参 考 文 献


[1] Parnas D L. On the criteria to be used in decomposing systems to modules[J]. Commun. ACM, 1972, 15(12): 1053-1058

[2] 何丽莉，金淳兆，冯铁，等. 关注点分离问题研究综述[J]. 计算机科学，2005，32(2)：129-132

[3] 曹东刚，梅宏. 面向 Aspect 的程序设计——一种新的编程范型[J]. 计算机科学，2003，30(9)：5-10

[4] Gregor K, Lamping J, Mendhekar A, et al. Aspect-oriented Programming[C]// Proceedings of the 11th European Conference on Object-Oriented Programming(ECOOP'97). Finland [S. l.]: Springler-Verlag, 1997: 220-242

[5] Filman R E, Friedman D P. Aspect-oriented programming is quantification and obliviousness[C]// Workshop on Advanced Separation of Concerns at OOPSLA, 2000

[6] Filman R E. What is AOP, revisited[C]// Workshop on Multi-Dimensional Separation of Concerns at ECOOP. 2001

[7] AspectJ home page[OL]. 2010. 11. http://eclipse.org/aspectj/

[8] AspectWerkz home page[OL]. 2010. 11. http://aspectwerkz.codehaus.org/

[9] JBoss home page[OL]. 2010. 11. http://jboss.org/jbossaop/

[10] Spring home page[OL]. 2010. 11. http://www.springsource.org/

[11] JAC home page[OL]. 2010. 11. http://ostatic.com/jac-aop

[12] AspectC++ home page[OL]. 2010. 11. http://www.aspectc.org/

[13] AspectDotNet home page[OL]. 2010. 11. http://aspectdotnet.codeplex.com/

[14] AspectC home page[OL]. 2010. 11. http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

[15] Steimann F. The Paradoxical Success of Aspect-Oriented Programming[C]// International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). 2006

[16] Kersten M. Comparison of the leading AOP tools[C]// Aspect-Oriented Software Development, AOSD'05. Industry track, Invited talk

[17] 唐祖锴,彭智勇. 面向方面程序设计语言研究综述[J]. 计算机科学与探索,2010,4(1):1-19

(上接第 95 页)

$(L^B:(pol))(s/vs,a/va,o/vo) \equiv L^{B(s/vs,a/va,o/vo)}:(pol(s/vs,a/va,o/vo))$

$(\Delta:(pol))(s/vs,a/va,o/vo) \equiv \Delta(s/vs,a/va,o/vo):(pol(s/vs,a/va,o/vo))$

$(\delta:(pol))(s/vs,a/va,o/vo) \equiv pol(\delta_s(s)/vs,\delta_a(a)/va,\delta_o(o)/vo)$

下面以一个例子来说明在本文策略逻辑框架下的策略评估。在 3.2 节中,介绍了医疗组织 $A$ 和 $B$ 采用第 3 种合成方式合并各自策略后得到的结果为 $\delta_A:(pol_A)\wedge\delta_B:(pol_B)$。现在假设有 2 个访问请求:((3, \$6),(read),(*medical_info*, 1))和((4, \$5),(read),(*medical_info*,0)),代入合并结果后得到

第一个访问请求:

$(3>2\wedge \$6-\$1\geqslant \$5)\wedge(\text{read}=\text{read})\wedge$

$(medical\_info = medical\_info\wedge 1<2)\wedge$

$(3>2\wedge \$6+\$1\geqslant \$7)\wedge(\text{read}=\text{read})\wedge$

$(medical\_info = medical\_info\wedge 1<2)\equiv T$

第二个访问请求:

$(4>2\wedge \$5-\$1\geqslant \$5)\wedge(\text{read}=\text{read})\wedge$

$(medical\_info = medical\_info\wedge 0<2)\wedge$

$(4>2\wedge \$5+\$1\geqslant \$7)\wedge(\text{read}=\text{read})\wedge$

$(medical\_info = medical\_info\wedge 0<2)\equiv F$

由此可见,上述复合策略允许第一个请求,第二个则被拒绝。

**结束语** 在大规模分布式多域应用环境中,基于属性的访问控制策略合并是一个值得研究的问题,该问题有着实际的应用背景。针对该问题,本文进行了相关研究,在将策略表述成为逻辑表达式的基础上,提出了一种逻辑融合框架,此框架不仅能够描述已有工作所支持的各种策略合并场景,还可以描述涉及属性值计算以及动态的策略合成方式。在逻辑框架基础上,又提出了一个推理系统,使得能够对合成策略是否满足合并方的期望进行形式化的验证。同时给出了评估本文所描述策略的方法,为本文所提方法应用于实际情况提供了基础。

策略合并问题涉及到许多方面的子问题。本文只是针对基于属性的访问控制策略在合成算子和框架上进行了一定的探讨。实际上,在实际应用中,策略制定者有时会希望在多个子域合并时,系统能够根据各自的子策略进行自动的合并,这就涉及到一个如何选择合并算子的问题。这方面的问题,也将是我们后续的一个研究方向。另外,在现实情况中,策略合并方式千变万化,我们将进一步分析各种可能存在的合成方式,并根据实际情况进一步扩展本文所提出的逻辑融合框架。

## 参 考 文 献

[1] Lee A J, Boyer J P, Olson L E. Defeasible Security Policy Composition for Web Services[C]// Formal Methods in Software Engineering (FMSE '06). ACM, 2006: 45-54

[2] Srivatsa M, Iyengar A, Mikalsen T, et al. An Access Control System for Web Service Compositions[C]// Proc. of IEEE International Conference on Web Services. 2007: 1-8

[3] Charfi A, Mezini M. Using Aspects for Security Engineering of Web Service Compositions[C]// Proc. of IEEE International Conference on Web Services. 2005: 59-66

[4] Ferraiolo D F, Kuhn D R, Chandramouli R. Role-Based Access Control (2nd Edition)[M]. Norwood, MA, USA: Artech House, Inc., 2003

[5] Moses T. QASIS extensible access control markup language (XACML) version 2.0 [S]. OASIS Specification, OASIS, 2005

[6] Ferraiolo D F, Gavrila S, Hu V C, et al. Composing and combining policies under the policies under the policy machine[C]// Proc. of the 10th ACM Symp. on Access Control Models and Technologies (SACMAT '05). ACM Press, 2005: 11-20

[7] Hu V C, Ferraiolo D F, Scarfone K. Access control policy combinations for the grid using the policy machine[C]// Proc. of the 7th IEEE Int'l Symp. on Cluster Computing and the Grid (CCGRID '07). Washington: IEEE Computer Society, 2007: 225-232

[8] Bonatti P, di Milano U, Vimercati S de Capitani di, et al. An algebra for composing access control policies[J]. ACM Trans. on Information and System Security, 2002, 5(1): 1-35

[9] 林莉,怀进鹏,李先贤. 基于属性的访问控制策略合成代数[J]. 软件学报,2009,20(2):404-414