维普资讯 http://www.cqvip.com

结合 AOP 与反射机制动态改变软件的行为*)

王小民 杨志辉 张 雄 许满武
(南京大学软件新技术国家重点实验室 南京 210093) (南京大学计算机系 南京 210093)

摘 要 软件系统的运行环境日益复杂,这样的复杂性已经远远超过了人的控制能力。面向对象的程序设计方法会造成关注点不能分离,代码纠缠在一起,使得软件的模块性与复用性大大降低。面向方面的程序设计(Aspect-oriented programming, AOP)[5]可以很好地分离关注点使软件更好地模块化。使用反射机制(Reflection),可以使程序在运行时通过自省(introspection)了解自己的状态,自己调节(intercession)自己(运行时自动修改程序),即动态地获得新的行为的能力。我们结合使用这两种方法的优点,使用 AspectJ 和 Java 的反射机制使得软件在运行时可以根据运行情况动态地改变行为。

关键词 面向方面,反射,关注点,动态改变,软件

Combining Aspect Oriented Programming and Reflection to Implement Software Adaptation Dynamically

WANG Xiao-Min YANG Zhi-Hui ZHANG Xiong XU Man-Wu
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093)
(Department of Computer Science and Technology, Nanjing University, Nanjing 210093)

Abstract The environment complexity of software continues to increase; it's beyond the ability of human to control it. Object-oriented programming could result in crosscutting concerns and code tangling; it is difficult in modularity and reusability. Aspect-oriented programming is an instrument for separation of concerns and enables modularization of crosscutting concerns. Reflection, the technique that permits a program to inquire about its own state at run time (called introspection), and permits modification and adaptation of the run time software. We combine virtue of AOP and reflection, and use AspectJ and reflection of Java to implement software modify its behaviors and respond to changing conditions during execution.

Keywords Aspect-oriented, Reflection, Concerns, Dynamic adaptation, Software

1 引言

计算机的出现就是为了帮助人们自动、快速地计算需要解决的问题。为了更加方便、快捷地开发出我们需要的软件,出现了软件开发的概念。软件的开发过程是人类将现实世界中的问题抽象化,用计算机进行解决的过程。人们在寻求解决现实世界中遇到的问题时,先后出现了很多种解决问题的办法。开始用面向过程的方法解决问题,后来为了实现软件的好的结构性和复用性,并且能够更好地抽象现实世界中的问题,出现了面向对象的软件开发方法。这种方法是将现实世界中的问题抽象为一个个的类,通过类的对象之间的交互去描述问题。它可以很好地抽象出现实世界中的问题,进而可以很方便地解决。

面向对象设计最根本的思想是,将真实世界领域中的实体及各自的行为抽象为对象。但是由于程序的运行就是一堆对象的相互交互,一个对象在向另一个对象发送消息时,常常会有功能性(functional)的代码和非功能性(nonfunctional)的代码纠缠(tangling)在一起,不能很好地分割,所以面向对象方式设计的缺点是各个对象混合的属性和一些非功能性代码的重复而使得代码混乱和不利于复用。我们将在本文中介绍使用 AOP 的方法来解决这个问题。

通常人们都集中于研究软件的静态行为,比如软件工程、软件的设计方法等等。然而,软件系统在运行时各软件模块和组件之间的交互关系与设计时的关系完全不同,而且软件在运行时的系统环境也十分复杂,这在软件设计时是无法考虑周全的。随着现在应用软件的规模越来越庞大,运行环境也日益复杂。普适计算(Pervasive Computing)概念的提出更加促进软件系统的复杂性的增加,已经远远超过人的控制能力。而且越来越多的软件系统需要长时间地运行,以提供服务。这些软件系统在运行的过程中由于运行环境的变化常常会出现软件老化(Software Aging)的现象,比如性能降低、产生错误信息等等。这一现象对一些需要长时间运行或任务至上(Mission-critical)的系统来说是要避免的。在其运行过程中,由于运行环境的改变或者软件本身的缺陷,需要改变软件的行为。而且为了保持软件系统的可靠性和可用性,这种行为的改变是动态的。所以,要更好地解决软件老化或软件崩溃的问题,必须更加注重软件系统的运行时刻,也就是软件的动态行为。

软件容错技术是现在比较有效的解决软件老化与软件崩溃的方法。软件容错的核心是将一个大的软件系统分割成一个个模块,每个模块是一个服务单元,其中一个模块出现问题并不会影响其他模块。但是由于软件设计时组件之间的紧密

*)本课题得到中国博士后科学基金、江苏省博士后科研资助计划资助。王小民 硕士研究生,主要研究方向:软件方法学、自主计算;杨志辉 硕士研究生,主要研究方向:自主计算、计算系统性能保持;张 雄 硕士研究生,主要研究方向:软件方法学、计算系统性能保持;许满武 教授,主要研究方向:数理逻辑、软件方法学、自主计算。

维普资讯 http://www.cqvip.com

耦合问题,常常会出现一个模块的变化不得已会改变其他模块的情况。为了使得代码更加简洁而且复用性更高,必须解决紧密耦合的问题。而且,容错系统也有其他的一些缺点。首先,容错系统需要更加复杂的管理,比如复制的组件的版本问题。其次,要维护容错系统,使之正常正确地运行,对用户提供系统的一致性还很有难度。为此对软件的动态自适应能力的需求越来越强烈。它要求软件在运行过程中能够根据运行环境的情况动态地改变行为。本文将就这一问题使用面向方面的程序设计方法(Aspect oriented programming, AOP)结合反射机制实现关注点分离并达到动态改变软件行为的能力。

本文结构如下:第2部分介绍AOP与反射的一些基本概念和优点,并给出它们之间的关系。第3部分基于一个实例来了解如何结合AOP与反射机制来实现程序关注点的分离与动态的改变软件的行为。最后是结论。

2 AOP与反射

2.1 基本概念

由于软件的设计不仅是完成功能性模块的设计,还要完成非功能性模块的设计,如日志记录和安全性检测等。当这些非功能性的代码分散在功能性的模块中时,会导致功能性的模块中重复地包含这些非功能性的代码,所以面向对象的系统通常会导致出现一些难以修改的、代码不可复用的类。面向方面的概念被人们由此提出。面向方面的程序设计(AOP)是一种新的编程技术,这一概念最早是由Gregor Kiczales等人提出。从1984年至1999年,Kiczales在Xerox Palo Alto研究中心(PARC)开展了AOP方面的工作,并且作为开发领导者对其进行了实现。

现在使用的著名的AOP工具主要有两种,分别是AspectJ和AspectC++。AspectJ是对Java面向方面程序设计的实现,是对Java语言的扩展。它只是对Java增加了很少的一些结构:切入点(pointcut)、通知(advice)、内部类型声明(inter-type declaration)和方面(aspect)。切入点和通知动态地影响程序的执行流,它是AspectJ的动态部分。内部类型声明允许程序员修改程序的静态结构,比如类的成员和类之间的继承关系。AspectC++是对C++语言的扩展,它和AspectJ中的语法差不多。

在面向对象的编程中,模块性的天然单位是类,横切关系是一种跨越多个类的关系。典型的横切关系包括日志记录、安全检测、性能优化等。通过将基本的设计重点从传统的面向对象编程(OOP)的层次结构中转移出来,AOP及其设计原理允许软件设计人员用一种和面向对象垂直而又互补的方式来考虑设计。面向方面的程序设计是一种将横切关注点(crosscutting concerns)模块化的方法。

AOP通过促进另一种模块性补充了面向对象的编程,该模块性将横切关系广泛分布的实现聚拢到一个单元。这种单元称为Aspect,这就是名称面向方面的编程的来历。通过划分Aspect代码,横切关系变得容易处理。AspectJ提供两种横切执行,分别是动态横切和静态横切。静态横切允许通过引入附加的方法和属性来修改对象的结构,而且利用静态横切可以整合第三方代码并解决紧密耦合的问题。而动态横切是可以在程序运行时在一个执行点定义程序的其他行为[12],是面向方面的程序设计的基础之一。

所谓方面(Aspect),"从抽象意义上讲,是对软件系统构件的性能和语法产生一定影响的一些属性;从设计上讲,是横切系统的一些软件系统级关注点;从实现上讲,是一种程序设计单元,它支持将横切系统的关注点封装在单独的模块单位中,是AOP将横切关注点局部化和模块化的实现机制"[4]。

使用AOP,我们需要一个在一个程序中识别连接点的方法。AspectJ使用切入点来描述一个或更多的连接点。切入点是一个用来描述一套连接点的表达式。可以把切入点想象成对你的编码的一个查询,用它来返回一系列的连接点[5]。

下面介绍一下AOP中的一些概念:

- 连接点(join point)是程序执行中精确执行点。
- 切入点(pointcut)声明了一组程序执行点,比如函数的调用。它是一个用于捕捉连接点的结构。定义了程序动态行为产生的地点,除了捕捉连接点之外不做任何事情。
- 通知(advice)是切入点的可执行代码。通知有在连接点之前(Before)、连接点之后(After)和周围(Around)执行三种类型。当程序运行到连接点时,通知便被触发运行。方面中的通知类似于类中的方法,可以访问类中的所有成员[12]。

方面(aspect)类似于Java编程语言中的类。方面定义切入点和通知,并由诸如AspectJ这样的方面编译器来编译,以便将横切织入(weaving)到现有的对象中。

AspectJ通过对Java进行扩展,并提供了编织(weaving)规则。其中切入点和通知指定了织入规则。方面中包含切入点和通知。连接点、切入点和通知关注软件的动态属性,而通知改变了程序编码的运行特性。

当编写了方面(Aspect)代码后,是通过将方面代码织入到源代码中执行的。那么如何将方面代码织入到源程序中呢?一般有两种编织方式,分别是静态织入和动态织入。静态织入方式是在源代码级织入方面代码,AspectJ就是用的这种方式。而动态织入是在程序的运行阶段织入方面的代码。由于它容许程序的设计人员可以在软件运行时动态地加入和移除方面[13],所以它的特点是灵活性强。

反射机制是计算系统感知自己和控制自己行为的能力,而且它能够根据条件的改变调整自己的行为[1]。类似于人类的神经系统,可以自我感知自己的环境和状态。在软件系统运行时,我们可以使用反射机制根据软件的运行环境来调节自己的行为,从而可以得到灵活的、可自适应的软件系统。

java.lang.reflect包提供了Java类反射机制,比如一个Java程序能够请求得知一个对象所属的类,查询类中的方法,而且调用其中的方法。访问反射数据的过程常常被称为"具体化"(reification)[2]。Java中的Class和Method类是元类(metaclasses),它们的对象是元对象(metaobjects)。利用反射包中的类不仅可以对类型进行详细检查,而且可以在程序运行时动态地加载类,并使用这个类创建该类的实例,从而可与这些动态创建的对象进行交互,并且通过反射机制来调用其方法。

2.2 关系

AOP的主要作用是分离关注点,将非功能性的代码封装在方面(Aspect)中,在程序运行时将方面代码织入到源代码(非方面代码)中。方面中的切入点和通知指定了织入规则。

反射系统分为两个层次:元层(meta level)和基层(base level)。元层主要提供系统内部的相关信息,包括类型、参数和方法调用等信息,它是有关计算的计算(computation about computation)[8]。基层构建于元层之上,是软件的基本行为。我们可以通过元层提供的信息和操纵方法来动态改变基层的行为。元对象(meta-objects)通过元对象协议(metaobject

protocol，MOP)来控制基层对象。在反射系统中，基层对象受到元对象的控制，它们之间通过元链接来实现[6]。

与AOP中连接点的概念类似，反射系统通过元链接使用称为钩子(hooks)的概念来实现从基层代码转向到元层代码。它们的共同点都是在基层代码的某一处触发这种执行的转向。而从分离关注点的角度来说，AOP与反射系统都具有这个能力，只是实现的方法和角度不同。如图1所示。

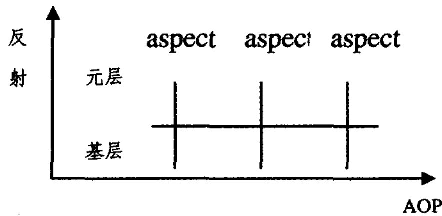

图1 AOP与反射在分离关注点方面的关系

反射机制具有较好的动态特性，而AOP则根据对AOP不同的实现表现出不同的特性。AspectJ是基于编译时刻对程序代码的修改，是静态的织入方式，而PROSE平台则支持动态的AOP[9,10]，使软件的运行过程中能够动态地织入或移除方面，是动态的织入方式。

3 实例

3.1 分离关注点

前面提到，面向对象的程序设计常常会造成代码的混乱和交叉。这种混乱的代码十分难以维护。而面向方面的程序设计给我们提供了消除混乱代码，将横切关注点模块化的方法，从而将冗余的代码封装进方面中，并且它提供了在方法执行前检查和执行后对数据进行处理的机制。比如，我们可以使用before通知在对方法调用前检查其合法性，从而增加交互的安全性。

下面我们用一个例子来说明问题。假设有一个银行账户类(accountBase)对客户提供取钱和存钱业务，客户类(clientBase)对象和银行账户类对象通过方法调用来实现交互，这两个类都属于基层部分。部分代码见图2和图3。

```
Public class clientBase {
  Public static void main (String [] args) {
     accountBase obBase = new accountBase();
     obBase.withdraw (****);
     }
  }
```

图2 客户类的代码

客户类对象要调用银行账户类中取钱或者存钱方法。图2中只显示了客户类的对象调用银行账户类中的withdraw()方法。银行账户类提供了两个接口，分别是取钱(withdraw)和存钱(deposit)方法供客户类对象调用。其中静态变量account用来表示银行中客户的存款余额。我们通过客户取的钱数即方法调用时的参数来模拟程序的运行环境。

如图2所示用银行账户类对象obBase调用withdraw方法时的参数来表示，以＊＊＊＊表示。如果这个参数小于或等于存款余额，则执行基层accountBase中的方法，而如果取的钱数大于存款余额，希望根据这个环境信息来执行其他动作。

```
Public class accountBase {
  Static int account = 10000;
  Public void withdraw( int wmoney ){
     //用户身份验证
       ......
     account = account - wmoney;
     System.out.println("in accountBase!");
     System.out.println("now account
        is"+account);
     }
  Public void deposit( int dmoney ){
         //用户身份验证
       ......
     acount = account + dmoney;
     System.out.println("in accountBase!");
  }
  }
```

图3 银行账户类的代码

这个交互过程可以用图4来描述。图中的方框表示，在客户类对象调用银行账户类的withdraw和deposit方法之前都进行了客户的身份验证工作。方框表示的就是这部分冗余代码。这种非功能性的代码穿插在源代码中，造成了代码的冗余。如果源程序比较庞大，则这种混乱的代码很难维护，也很难实现复用。

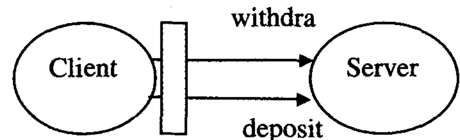

图4 客户类对象与银行类对象交互图

我们的目的是在类对象之间交互时消除混乱代码，实现关注点的分离。我们使用AOP的概念将非功能性代码封装到方面中以达到关注点的分离。前面提到了连接点和切入点的概念，利用这两个概念，在方面中定义了一个切入点accountdrawcalls，它用来捕捉调用银行账户类中的withdraw或deposit方法的行为，如图5所示。

```
Call(void accountBase.withdraw(int)) &&
args(money)
```

这里指定这个调用无返回值，参数为int类型。另外指定args(money)，由此可以得到连接点的上下文环境信息。在这里我们得到了调用这些方法的参数money。这些上下文环境信息对在这些切入点织入新的代码是十分有用的。

另外可以看到，将前面代码中withdraw和deposit方法中用户身份验证的代码写在了before通知中。这样银行账户类中关于身份验证的代码都不见了，极大地实现了关注点的分离。并且在源代码中只存在功能性的方法，实现了功能性代码和非功能性代码的分离。

使用AOP方法进行身份验证和在源程序中写入身份验证的代码相比较有如下好处：

- 只需要在一个方面中写入身份验证的代码，减少了源代码中冗余的身份验证代码。
- 插入和删除身份验证的代码非常方便。
- 可以有一个重复使用的方面。

3.2 动态改变软件的行为

可以看到前面的实现方式是固定的，即各个类所提供的

维普资讯 http://www.cqvip.com

接口和实现都是定义好的，不能在程序的运行过程中根据一些条件的判断来动态地改变程序的行为。我们希望能够在程序的运行时刻根据运行环境来判断是执行基层的程序还是改变程序的行为。我们通过在程序的运行过程中动态地加载一个类来实现给程序提供新的行为。我们称这个动态加载的类为代理类。

相比上面提到的 before 通知，around 通知具有更好的动态性。around 通知经常用于在连接点处替换执行原始方法或用新的参数调用原始方法，也可以多次执行原始方法，而且可以通过引进新的组件而改变程序的结构[7]。

我们的目的就是在程序的运行时刻动态地替换组件，根据运行环境的变化或某些条件而引起新的行为的产生。而这些行为在源代码中是没有考虑到的。这种软件被认为是动态自适应的[3]。

图 5 显示了部分的方面(Aspect)代码。在程序的运行时刻，在 around 通知中动态地加载进代理类 accountDelegate。并且实例化 accountDelegate 类，从而调用其相应的方法。由于通知的执行是在连接点触发的。由于在连接点处获取了程序执行时的参数，需要根据参数的情况来决定是否要执行新的组件的行为。我们希望用这种方法来模拟软件的运行环境的变化对软件行为的影响。

```
Public aspect validateAspect {
   Pointcut accountdrawcalls(int money)
    :( call( void accountBase.withdraw(int))
    || call( void accountBase.deposit(int)))
    && args(money);
     ......
   before(int money):
   accountdrawcalls(money){
     System.out.println("验证账户！");
     System.out.println("money");
   }
   void around(int money)
   :accountdrawcalls(money){
     myClassLoader objClassLoader
     = new myClassLoader();
     Class clas = objClassLoader.loadClass
     ( "accountDelegate" );
     Object obClas = clas.newInstance();
    //得到装载类的实例
     Class paraType [] = {Integer.TYPE};
     Method metaFunction = clas.getMethod
     ("withdraw", paraType);
     Integer funargs = new Integer(10000);
     Object arg [] = {funargs};
     if ( money > accountBase.account )
        metaFunction.invoke (obClas, arg);
     else
        proceed(money);
       //如果参数合法，继续执行基类的方法
    }
```

图 5　部分方面代码

如果 clientBase 对象调用 accountBase 类中 withdraw 方法的参数 money 大于银行余额 account，则通过自己定义的类加载器(限于篇幅，详细代码没有列出)动态地加载进代理类 accountDelegate，并且反射调用代理类中的方法。myClassLoader 类就是自己定义的类加载器，它继承于 Java API 中 ClassLoader 类，负责在程序的运行时刻查找和加载类。就是使用这个自己定义的类加载器，我们可以加载进自己需要的组件，动态生成被引用的类。

由于程序是在运行时刻才根据程序的上下文环境来判断是否加载进新的组件，执行新的行为。这是软件事先并不知道的。这种动态的 Java 代码可以快速响应环境变化的要求。它可以被用来实现真实的动态服务和时时改变的业务规则。图 6 显示了执行流程。

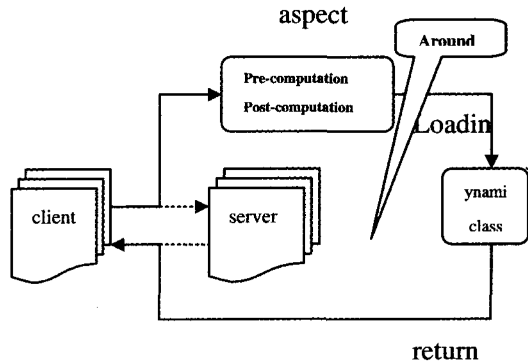

图 6　执行流程

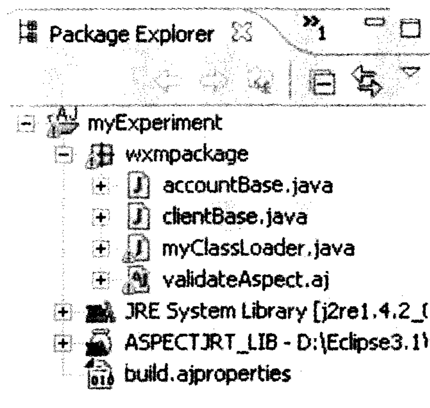

图 7　工程结构

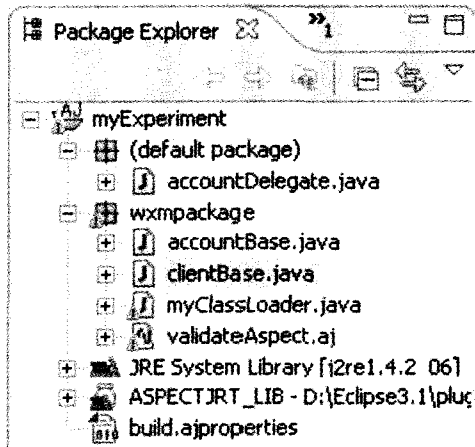

图 8　工程结构

在加载进代理类 accountDelegate 之前，工程结构如图 7 所示。当在方面代码中由于某些条件的满足而触发执行新的代码时，动态地加载进代理类，使得程序改变自己的行为。这时的工程结构图为图 8 所示。可见在程序的运行过程中，我们在方面代码中获得了基层程序的执行环境，从而触发动态地加载新的组件，并且使得基层代码改变自己的行为去执行代理类的方法。

总结　本文指出了 AOP 与反射编程的关系，并且提出

维普资讯 http://www.cqvip.com

了利用面向方面的编程与反射机制结合的方法来实现在软件在运行中动态改变软件的行为。AOP 通过引入方面这种模块化单元,使得程序关注点得到分离,消除了混乱代码。反射机制提供了一种使计算系统感知自己和控制自己行为的能力,利用反射机制这种可以自省的能力可以对自己的行为和环境进行判断,结合 AOP 的概念可以动态地加载新的组件以达到改变自己行为的目的。但是由于元信息的存储和管理使得反射机制带来的性能损耗比较大,并且 AspectJ 用的是静态织入方式。然而,动态 AOP 语言已经出现并且已经得到应用。它的优点是使得软件在运行时可以修改方面并且可以动态地织入新的方面[11],而且程序的设计人员可以在软件运行时动态地加入和移除方面,特点是灵活性强。我们下一步要研究动态 AOP 的应用。

参考文献

1 Maes P. Concepts and experiments in computational reflection. In:Meyrowitz N. editor. In:Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA'87),Orlando,FL,USA,ACM Press, October 1987. 147~155
2 Sullivan G T. Aspect-oriented Programming Using Reflection and Metaobject Protocols. Communications of the ACM, October 2001, 44(10):95~97
3 Yang Z, Cheng B H C, Stirewalt R E K, et al. An Aspect-oriented Approach to Dynamic Adaptation. ACM, 2002. 85~90
4 Cheng H. Aspect oriented programming methodology. Information Technology Letter, 2005, 3(1):1~4
5 Kiczales G, et al. Aspect-Oriented Programming. In: proceedings of ECOOP'97. LNCS 1241, Springer-Verlag, June 1997
6 Pierre-Charles D, Ledoux T, Bouraqadi-Saadani N M N. Two-step weaving with reflection Using AspectJ. In: OOP-SLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, October 2001
7 Bencomo N, Blair G, Coulson G, et al. Reflection and Aspects meet again: Runtime Reflective Mechanisms for Dynamic Aspects. In: ACM International Conference Proceeding Series, Vol118,2005
8 Ebraert P, Tourwe T. A Reflective Approach to Dynamic Software Evolution. In: the Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution(RAM-SE'04) in conjunction with the 18th European Conference on Objected-Oriented Programming(ECOOP), 2004
9 Nicoara A, Alonso G. Dynamic AOP with PROSE, In: Proc. of International Workshop on Adaptive and Self-Managing Enterprise Applications(ASMEA'05) in CAISE'05,Portugal,2005
10 Popovici A, Gross T, Alonso G. Dynamic weaving for aspect-oriented programming. In: Proceedings of the 1st international conference on Aspect-oriented software development, Enschede,The Netherlands, April,2002
11 Greenwood P, Blair L. Using Dynamic Aspect-oriented Programming to Implement an Autonomic System. In:Proceeding of the 2004 Dynamic Aspects Workshop(DAW04),Lancaster, RIACS, 2004. 76~88
12 Kiczales G, Hilsdale E, Hugunin J. et al. An overview of AspectJ. Lecture Notes in Computer Science, 2001, 2072:327~355
13 Sato Y, Chiba S, Tatsubori M. A Selective, Just-In-Time Aspect Weaver. In: Second International Conference on Generative Programming and Component Engineering(GPCE'03), Erfurt Germany, 2003. 189~208

(上接第 240 页)

工分析。

4 语用层分析

语用层分析是体系结构最高层次的分析,与语法层和语义层分析不同,语法层和语义层分析主要针对体系结构静态特征和非效用的特征分析。语用层分析主要对体系结构整体效用的分析,注重的是体系结构对应系统的整体功效,分析体系结构对应系统的效用以及对需求的满足程度。通常,体系结构的语用层分析必须结合具体背景,在背景环境下分析系统的性能、效能。

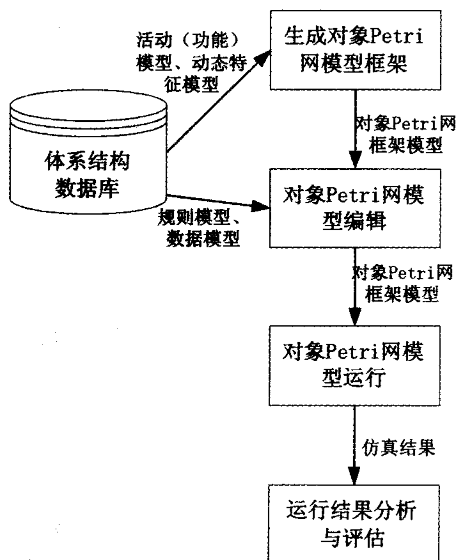

图 2 基于可执行模型的语用层分析过程

语用层分析最典型的方法就是采用基于可执行模型的方法,通过对可执行模型的仿真,对体系结构对应系统的性能、效能进行定量的分析与评估。

根据体系结构产品的特征,考虑采用对象 Petri 网[7]作为可执行模型,将体系结构产品转变为对象 Petri 网模型的语用层分析过程如图 2 所示。

首先,利用体系结构数据库中作战活动模型(或系统功能模型)、状态转换模型等信息,构建对象 Petri 网模型框架,形成 Petri 网模型的基本结构;再利用体系结构数据库中规则模型、数据模型以及性能参数模型,编辑对象 Petri 模型中转移的动作函数、谓词函数、时间函数等,生成完整、可执行的对象 Petri 网模型,并利用对象 Petri 网仿真平台仿真模型,跟踪执行过程和收集运行结果,分析体系结构对应系统的性能和效能。

此外,还可以采用传统的专家打分的方法,进行语用层的定性分析。

结束语 针对 C4ISR 体系结构的设计问题,提出体系结构全信息分析模型,并研究了语法层、语义层、语用层分析的内容和常用方法。在实际分析中,各层析分析原则和方法还有待进一步完善和补充。此外,语法层和语义层的分析功能和算法可以集成到体系结构设计支持工具中;保证体系结构在设计的同时就完成相关的分析和检验工作。

参考文献

1 C4ISR Architecture Working Group. C4ISR Architecture Framework Version 2.0[R]. U S: Department of Defense, 1997
2 DoD Architecture Framework Working Group. DoD Architecture Framework Version 1.0 (Draft)[R]. U S: Department of Defense, 2003
3 Levis A H, Wagenhals L W. C4ISR Architecture Framework Implementation[EB/OL]. http://viking.gmu.edu/http/AFCEA-503Y/CourseBook/index.html, 2002-09-30
4 Kazman R, Klein M, et al. The Architecture Tradeoff Analysis Method[C]. In:Proceedings of ICECCS, Monterey, CA, 1998
5 钟义信. 信息科学原理[M]. 北京:北京邮电出版社, 1996
6 Popkin Software Corporation. Popkin Software's News & Events [EB/OL]. http://www.popkin.com/newsandevents/newsandevents.htm, 2002-11-01
7 国防科技大学 C3I 研究中心. 基于对象的 Petri 网建模仿真环境技术报告[R]. 长沙:国防科技大学, 1999