

基于构件描述的 动态更新方法

复旦大学计算机科学系软件工程实验室 袁磊 张海飞 钱乐秋

摘要 本文对构件描述语言(CDL)作了介绍,以及在动态更新中的初步应用:将系统分解为原子形式(构件),对其构件及构件内部之间关系、内部与外界接口作精细的描述,软件的更新很大部分是对构件的可替换部分作修改,与外界接口是相对固定的。在动态的条件下,将待修改的构件覆盖以新版本构件,及其具体过程。

关键词 CDL ADL 软件重用 IDL CORBA SOFA

引言

为了提高软件的生产率,基于构件的软件工程(CBSE)方法,UML,构架描述语言(ADL)等工具便得到了发展,因为基于构件的软件开发方法一定程度上解决或实现了软件的重用(Software Reuse)和软件即插即用(Software Plug & Play)。软件的更新和演化是软件工程的重要部分,本文试图通过构件的描述对基于构件的软件动态更新作初步的讨论。

随着软件技术的更新,很多旧有的应用系统都面临着日益落后的问题,或者是因为速度问题,或者是因为界面问题,等等,用户觉得有必要对之进行更新。对于这些系统,可以不按照软件工程的传统方法(需求分析,快速原型……)进行。即我们可以使用一定的构架描述语言(Archintecture Description Language)将构架描述出来,这对于描述构件之间的关联及功能组成是相当重要的。再对构件可替换部分做适当的修改,完成构件的动态更新。在本文中,我们将结合一个实例来具体说明。

一个例子

从1999年10月起,我们受中国邮电器材总公司华东分公司经营某部的委托,对其原有的企业局域网及业务系统进行更新和扩充。其系统构架(部分)中各个模块关系如图1:

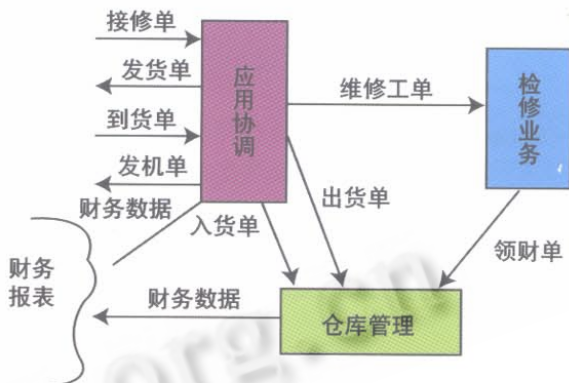


图 1

本文中仅讨论检修业务这一模块,因为这一部分逻辑较易用既定的语言来描述。

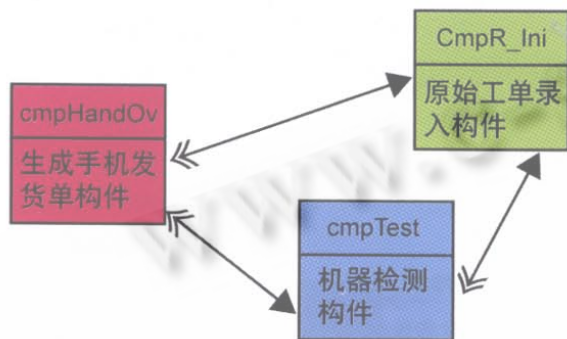
构架描述的概貌

我们知道,构架描述语言的基本实体是构件(Component),接口(Interface)和连接器(Connector)。构件是一个计算单元,有自己的计算能力及数据。几乎每个构件都需要其他构件的服务;同时,经过计算和处理为另外的构件提供服务。构件和外界的交互点即为构件界面。但是描述这些接口却有困难。许多ADL都支持构件接口的描述,但是采用的方法却大相径庭。

在本文中,我们使用的是较常用的描述方法:构件描述包括一族逻辑组成,性能要求和他们之间的约定,即构

件之间是互为提供者和需求者的关系,可以看作是一定的约束关系。只要对服务的接口描述不变,那么对构件实现动态更新是可行的,我们只须将构件的可替换部分作相应的修改即可。

基于提供/要求服务的系统的语义描述:一个具有出口和入口的黑盒框架,即需要预知既定构件提供的确切功能及要求得到的确切服务。构件可分为基本构件和合成构件。基本构件的描述包括提供和需要的服务(接口),此时其内部结构是透明的。合成构件的描述是一组构件的集合(子构件也可能是合成的),以及它们之间的绑定关系和服务依赖关系。一个例子见图2。



注 → 服务提供:
→ 服务要求:

图 2

构件模板

基于以上的分析,我们得以对构件的结构作抽象的描述。这里我们采用OMG提供的IDL来描述服务,因为IDL本身提供了很强的描述接口的功能,通过它可以将动态更新集成为CORBA构件模型,它是独立的,即能够在不同的语言上实现

构件模板的头部由两个属性来标识:

提供者姓名: 可能是商标

构件名称: “提供者+构件”可以唯一标识构件

构件的主体包括两部分: 提供的服务和要求的服务。

例如: 维修业务构件

```
#include Component_Bravo.lib
```

```
typedef cmpRepair component {
```

```
provider:ZY_Fudan;
```

```
title:Repair_bravo //bravo 是系统的名称
```

```
provision:
```

```
// 提供的服务, 见后面的定义和实例  
// 即对原始工单的对象进行维修和处理  
.....
```

```
requirement:
```

```
// 要求的服务
```

```
// 需要对待修机器的属性进行初始化工作
```

```
//.....
```

```
Other_Properties_Methods
```

```
}
```

1. 构件提供的服务

构件提供的服务指外界从它可以获得的服务,这必需通过外包(wrapper,即为待定的服务访问接口)。因而CDL应该能够生成正确的外包。一般来说,外包只在初始的接口处重新向导来到的调用,并且返回值,所以必需将输出参数和结果传送给正确的外包对象。下面是一个外包的名为Look_Repair_Bravo的方法:

```
public procedure Look_Repair_Bravo  
(pl,pt_Type,.....) {  
    Chick_in();  
    // 对第一次检查通过的手机  
    Goal(p1,p2,.....);  
    // 执行主要的检修功能  
    // 若有必要则向外包传送输出参数  
    Check_out();  
    // 确保输出的状态  
}
```

构件提供服务的描述以“provides”为关键字,下面是<接口类型,提供服务的名称>的序列。借用“public”,“private”,“protected”等词,可以这样标识:

```
#import "interfaces.CDL"  
typedef InterfaceC [$NumC] TprovC;  
template ProviderName C {  
    provides:  
    InterfaceA provA;  
    InterfaceB [10] provB;  
    TprovC provC;  
};
```

模板的头部包括构件名(C)和提供者的姓名。第一行表示一个类型为InterfaceA的外包。第二行申明10个类型为interfaceB的外包。最后一行,是构件的“动态”描述,它的大小在创建时由“NumC”决定。

2. 构件需要的服务

一个构件不可能自行实现其需要的所有服务,必需向其他的构件申请。下面是这一部分的描述:

```
#import "interfaces.dcl"
template ProviderName C {
    requires:
        InterfaceA [5] reqA;
        InterfaceB reqB; };
```

表明该构件需要 InterfaceA 类型的服务 5 个, InterfaceB 类型的服务 1 个。注意需要服务的描述在模板体会多次出现,即提供服务的描述和需要服务的描述可能是混杂的。

3. 构件启动参数

构件在创建自身时需要一些参数。有两种方式描述这些参数:

直接在构件头部描述(和传统表示方法类似)

类属的参数(所有的参数放在一个对象中,在设计阶段是不确定的)

两种方法各有利弊。前者必需在设计时声明所有的参数及类型,当然还包括其子构件的。当嵌套程度较深时,这些申明让人不知所云。后者的优点即在于此,所以构件可以根据需要动态使用参数,缺点是设计者不能精确的知道哪些参数是必需的,可能这些信息对运行是很重要的。基于这些,我们采用折衷的方法,构件描述包含一个参数申明的部分,在设计时,只须确定它们的名字,它们的类型在运行时才确定。JAVA 可以实现动态的类型验证和发布,解决了主要的问题。所有的参数包含在一个 Tproperty 类型的对象中,并且是唯一命名的。运行时由构件生成器将他们的类型向子构件发布,如下:

```
architecture ProviderName Look_Repair_Bravo
version "1.0" {
    properties:
        NumOfSubComp;
        // 子构件的数目
        // 如机型校验、身份验证等
        NumOfInstA;
        // 实例的数目
};
```

注意: 这些参数对构件管理器是无用的,它们对构件生成器创建构件时才是必需的。但是构件生成器对外界是不可见的,所以这些参数的传递必需通过构件管理器。

在这里,构架的版本号是很重要的,它决定使用哪个版本的子构件来生成实例,因为在同一构件中允许不同版本的子构件共存。执行对象是通过接口来描述的,因为 CDL 不可能过多的描述执行细节,执行对象的数目是动态的。

动态更新 & 更新过程

由 CORBA 等组织发起的 SOFA(SOFTWARE Architecture)工程试图建立这样一种环境,使用户能够方便地获取软件产品。一个完整的软件包由一些构件组成,它们可以完成特定计算功能,而且可不依赖于其他构件而独立更新。

动态构件更新(DCUP)是 SOFA 工程的一部分。它允许某个构件在系统运行时动态得到更新,全部过程是相对独立的,即对其他构件是透明的。典型构件模型见图 3:

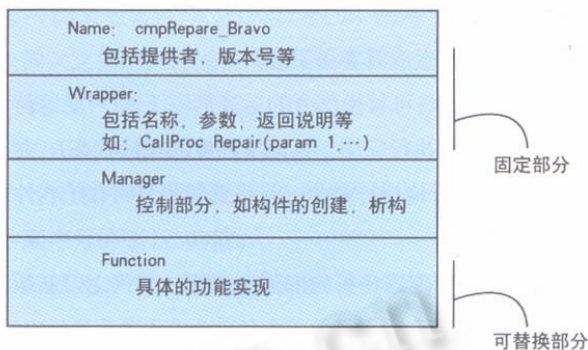


图 3

每一个构件包含一个管理器和生成器,可以拥有若干执行对象和子构件来协同完成功能。它提供的服务由执行对象来发布,这些对象不能直接访问,必须由特殊的对象—外包来完成,外包有和执行对象相同的接口。同时,构件还可以要求其他构件的服务。

构件分为控制部分和功能部分。生成器负责在完成初始化后创建构件,或者在更新消息到达时重建构件。所有服务的申请须通过管理器,它还有协调更新过程的作用。

另一方面,构件还可分为固定部分和可替换部分。在构件的生命周期中固定部分始终存在,包括管理器和外包;可替换部分仅存在于两次更新之间。

由上,构件的更新就是更新可替换部分。注意所有对内部对象的引用必须由外包再索引,但是在构件更新时,

外包是被锁住的。在运行时更新构件,其他构件是不知道的,大致情景是这样的(图4):

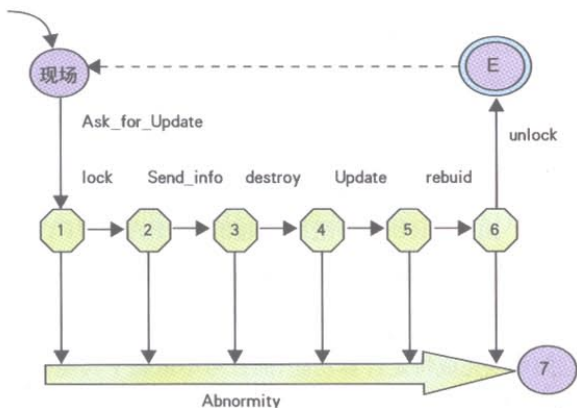


图 4

1. 构件管理器(可以通过代理Agent)锁住需更新的构件及与之紧密关联的构件的外包,并且将到来的调用进程挂起;
2. 向构件生成器发送消息;
3. 生成器将可替换部分的运算都停下来(可能等待它们将原子操作执行完毕,再记录相应的现场状态),并且把它们析构(析构执行对象,并将更新消息向其子构件循环发布);
4. 构件管理器将析构旧的生成器,并代之以新版本的生成器;
5. 新的生成器创建必需的内部对象(执行对象和子构件);
6. 完成之后,构件管理器把所有外包解锁;
7. 如果某个构件未收到消息,构件生成器将对它的子构件补发,并且继续计算。

从上面知道,每个构件都有控制部分,这部分代码是类似的(因为可以将构件的一些基本的方法抽象出来),所以可以通过某种描述自动生成。同样外包构件也仅在类型

和参数个数上有差异,调用的顺序都是相同的。基于上面的流程,类似于数据库中的锁操作,不必使整个系统停下来,只要暂时限制相关的构件就可以了。当然,还有许多实现上的细节问题,如:

如何加锁、解锁,对调用该构件的进程如何处理,和操作系统的接口,如何在网络环境中发布消息和保护现象……等等问题,都是非常重要的。本文主要从构件和构架的关系的角度来讨论,这些问题还有待于深入的研究。

小 结

文中从对构架的描述开始,由此将构件集成到系统中去,了解构件及其内部功能部分在整体中的结构和作用,和它们之间的通信关系、耦合程度等性质,将可替换部分表示出来。在对构件的描述时,我们采用了构件描述语言,它是基于ADL和IDL等描述语言推广的,进而可以利用这一工具来编写可更新的应用系统,由构件描述语言(CDL)编译器可生成易于更新的目标代码,无须关心过多的执行细节,应力求简洁性和可读性。目前,可动态更新构件框架已有JAVA版本,如JAVA CLASS LOADER可以实现动态下载对象。■

参考文献

- 1 Jason E.Robins, *Integrating Architecture Description Languages with a Standard Design Method*, Software Engineering Notes, 1997, No 3
- 2 V Menci, *Component Definition Language*, Master Thesis, Charlse University, 1998, 4
- 3 Jim Q.Ning, *Toward Software Plug-and-Play*, Software Engineering Notes, 1998, No 6
- 4 张海藩, 《软件工程导论》, 清华大学出版社, 1994
- 5 朱崇湘, 基于构件的软件工程化开发方法, 复旦大学硕士毕业论文, 1998