ARTEMIS-ARC 系统协同模型的自省式实现技术研究*)

马 骞 俞 春 马晓星 吕 建

(南京大学计算机软件新技术国家重点实验室 南京大学计算机软件研究所 南京 210093)

摘 要 基于运行时体系结构的协同模型能够为面向服务的协同应用系统的动态演化提供有效的支持。但是在实现层面上,如何使软件体系结构从抽象的规约转化为运行时实际的对象实体,并成为系统演化行为的直接载体,是一个较为困难的技术挑战。针对这个问题,本文提出一种基于计算自省的实现途径,主要包括基于面向对象程序设计语言构造的体系结构层面的元表示和元协议、基于体系结构上下文中对象引用重解释构建的因果互连机制,以及基于该因果互连机制的应用系统的动态重配置。以上实现方式在自行开发的服务协同系统 ARTEMIS-ARC 时得以实施。

关键词 软件服务,软件体系结构,动态演化,自省

A Reflective Implementation of the Coordination Model in ARCTEMIS-ARC

MA Qian YU Chun MA Xiao-Xing LU Jian

(State Key Laboratory for Novel Software Technology, Institute of Computer Software, Nanjing University, Nanjing 210093)

Abstract Runtime architecture-based coordination model provides a reasonable solution for dynamic evolution of service-oriented application systems. However, in implementing this coordination model it is a rather difficult challenge to reify the software architecture, which is commonly an abstract description, as a concrete object, which should act as the comprehensive and flexible carrier of evolution. Addressing this problem, an approach based on computational reflection is proposed. It involves designing a meta-representation and a meta-protocol in software architecture layer based on object-oriented mechanism, establishing a causal connection between the architecture object and the running system by re-interpreting object references in the architecture context, and supporting the dynamic reconfiguration of application system based on the causal connection.

Keywords Software service, Software architecture, Dynamic evolution, Reflection

1 引言

面向服务的计算(Service-Oriented Computing, SOC)是最近计算机产业界的研究热点之一。它提出了一种新的以软件服务为基本组成元素来构建应用程序的计算泛型,期望能够以一种快速灵活的、低成本的方式构建分布式的应用程序[1]。然而,目前已有的软件服务标准和运行支撑技术还不能完全满足 SOC 的要求,特别是现有的面向服务的协同应用系统在对开放动态的网络计算环境和不断演化的用户个性化需求的适应性动态演化方面亟待深入研究。

在我们自行开发的一个动态架构的软件服务集成平台 ARTEMIS-ARC[2]中,采用一种基于内置运行时体系结构的协同模型来刻画服务之间的协同关系,使其作为已有的服务协同标准的补充,从而为 SOC 环境下构建的应用系统引入动态演化的能力。具体来说,它从软件体系结构的视角来观察服务之间的协同行为,并将体系结构从抽象规约转变为运行时刻的具体的可操纵的对象实体,使服务之间的交互行为定义在该对象实体提供的上下文中。另一方面,系统运行中体系结构约束的(可能)破坏是系统演化的驱动因素,而由于服务之间的协同行为都将在运行时刻体系结构对象提供的上下文中解释,从而该对象可以直接作为系统演化行为的载体,其行为的变化可以驱动服务之间的协同行为的重新解释,完成动态重配置。

然而,要实现上述协同模型,我们面临如下两个技术挑战:1)如何将软件体系结构从抽象的规约具体化(reify)为面向对象程序设计语言中的对象,并允许外界对该对象进行操纵;2)如何确保该对象确实定义了运行时体系结构,即对该对象的操纵能够准确地反馈到系统本身,进而实现面向体系结构的动态演化。

本文针对上述问题,提出了基于运行时体系结构协同模型的一种基于计算自省的(Computational Reflection)实现途径。自省的系统,简言之,就是该系统应能够提供对自描述(称为元表示,meta-representation)和对自描述的操纵(称为元协议,meta-protocol),并且这种描述与系统的行为具有一种因果互连的关系(causal connection)[3]。我们认为,运行时体系结构对象实际上是从体系结构角度得出的一种元表示,而通过在体系结构对象中定义相应的方法来实现外界对该对象的操纵,构造出元协议。但是,我们强调,对体系结构对象的操纵必须满足一定的约束条件,从而确保系统体系结构的一致性。为此,我们在实现元协议时加入对体系结构风格的支持,以表达特定的系统组成和演化行为方面的约束,从而一定程度上保证对体系结构对象操纵的一致性。此外,我们采用了 Dynamic Proxy[4]的机制对系统组成部件的交互进行解耦合,从而将体系结构对象介入系统运行之中,使对象的引用能够在体系结构提供上下文中加以解释。本文还进一步探讨了自省式实现对应用系统的在线动态演化的支持,尤其是对非预设性演化的支持,从实现的角度解释了在线动态演化的可行性。

我们认为,上述的实现途径为支持应用系统的动态演化提供了一种一般性的解决方案。基于运行时体系结构的协同

*)国家自然科学基金(60403014,60273034)、863 计划(2005AA113160,2005AA113030)和 973 计划(2002CB312002)资助项目。马 骞 硕士生,研究方向为 Internet 软件技术、软件体系结构;俞 春 硕士生,研究方向为 Internet 软件技术、软件 Agent 技术;马晓星 博士,研究方向为 Internet 软件技术、软件体系结构;吕 建 博士,教授,博士生导师,研究方向为软件自动化、并行程序形式化方法、面向对象语言和环境。

模型能够从总体上刻画系统的特征，约束系统各部分的协同行为和动态演化。而基于计算自省的实现技术能够有效地解决体系结构层面和实际运行系统之间的语义关联问题，确保体系结构层确实定义了系统的运行时体系结构。

我们将在第 2 节中简要介绍 ARTEMIS-ARC 系统基于运行时体系结构的协同模型；第 3 节中介绍协同模型实现的核心机制；第 4 节中具体讨论协同模型的自省式实现；第 5 节探讨基于运行时体系结构的在线动态演化；最后小结全文并简单展望未来工作。

2 ARTEMIS-ARC 系统的协同模型简介

软件体系结构描述了软件系统的构件组成、构件之间的相互关系和连接方式所形成的系统结构配置信息和约束[5]。近年来，随着软件系统动态演化需求的凸现，软件体系结构在系统的运行阶段中的作用日益得到重视，其主要原因在于体系结构信息是系统动态演化的重要依据和关键约束，软件体系结构的改变可能是系统动态演化的驱动因素，并在软件体系结构刻画了系统层面的特性，从而能够一定程度上保证系统动态演化的一致性和完整性[6]。

以往的工作中存在的基于软件体系结构来实现系统动态演化的尝试，主要是采用外部控制的方式[7]，见图 1(左)所示，把从体系结构视角描述的系统信息通过探测器(Probe)从运行系统中抽取出来，然后将对体系结构信息的改变通过效应器(Effecter)反作用于运行系统。从某种程度上来看，这种方式对动态演化提供了一定的支持，但是很难为各种应用系统找到一个相对比较通用的探测器和效应器的设计方法。

从本质上来看，外部控制方式的困难在于软件体系结构仍然停留在抽象的规约上，并没有真正参与到系统组成部分的协同中，因而很难对系统的动态演化有实质性的帮助[8]。解决这个问题关键在于要真正将体系结构作为协同逻辑编程的核心。基于此，在我们的运行时刻软件体系结构中，采用内置体系结构的方式，将传统的软件体系结构从单纯的抽象规约转换为实际可以操纵的对象，使软件体系结构的作用直接渗透到系统运行中，如图 1(右)所示。这时软件体系结构作为在运行时刻存在的对象，和一般的对象一样，同样具备状态和行为。因此，系统结构方面的规约将在该对象的状态中得到表达，而系统动态演化将具体化为该对象运行时的修改行为。

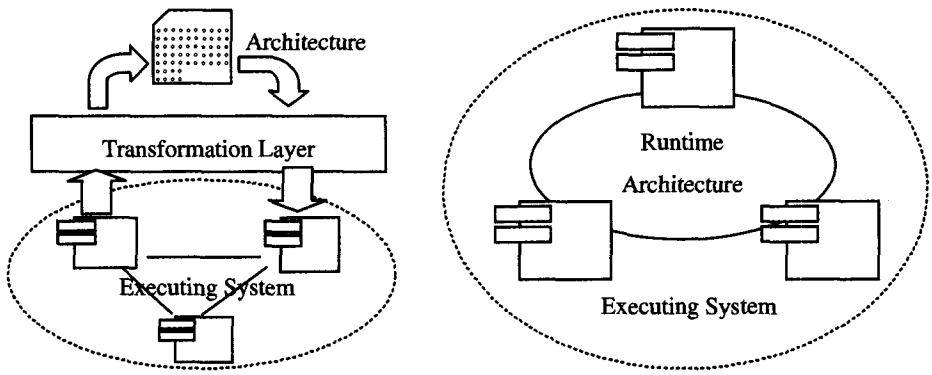

图 1　传统的外部控制方式(左)与内置运行时体系结构方式(右)

3 协同模型实现的核心机制 —— 自省

采用基于运行时体系结构的方式来解决系统的动态演化问题，本质是一种自省的途径。所谓一个系统具有自省的能力，从抽象的层面来说就是一个系统具有对自身状态和行为的推导和改变的能力。更为具体地说，自省的系统能够提供对自身的描述，并且这种描述与系统的行为具有一种因果互连的关系。因此，自省的系统具有两方面的特征：1)能够对自身的行为进行观察(Inspection)，产生自描述；2)能够动态演化(adaptation)，演化的根源在于自身描述的变化，这种变化与实际的运行系统之间具有本质的因果互连关系，从而能够确实反映到实际运行的系统中[3]。为此，自省的系统往往采用分层的结构来实现。基层(base-level)负责系统实际的计算功能。在基层的上面构建元层(meta-level)，完成对系统本身的计算。元层主要关心两个方面的问题：一是如何形成系统的自描述，又被称为元表示；二是如何向外界提供对元层进行操作的能力，又被称为元协议。在分层的基础上，更为重要的是要在两层之间建立本质的因果互连的关系，从而赋予实际运行系统动态演化的能力[9]。

在我们的协同模型中，体系结构层次实际上就构成了运行系统的元层，体系结构对象实际上是给出了一种系统的元表示，并且这种元表示强调了从一种全局的角度来观察实际的运行系统，避免了对系统整体的描述能力不足。在体系结构类中定义一些具体的方法，以完成外界对体系结构对象的操纵，从而定义出元协议。特别地，我们引入体系结构风格的概念来约束元协议的设计，特定的体系结构风格强调了构成体系结构的构件之间特定的组成关系，从而能够在一定程度上保证对体系结构对象操纵前后的一致性。

最后，在元层和基层之间的因果互连的建立上，我们发现，现在的软件服务的最终实现都是基于 EJB、DCOM 等面向对象技术的，所以在软件服务之间的交互大多数都是通过对象之间的引用关系来描述的。从更高的层面上来看，我们可以认为正是组成系统的各个部分之间的引用关系最终实现了系统的体系结构。因此，要在我们构造的体系结构对象和实际的运行系统之间建立本质的因果互连关系，关键也就在于要使系统各个部件之间的引用放在体系结构对象所提供的上下文中解释。也就是说，体系结构对象对实际应用系统的操纵应该通过在体系结构上下文中对部件之间的引用进行解释和重解释来实施。实际上，导致系统缺乏演化能力的根源就在于系统部件之间的“硬连线”，在系统运行时并没有从体系结构的角度去解释这些应用。例如，我们希望构建一个由旅行社和飞机场这两个基本服务构成的组合服务来完成查询飞机票价的功能，具体的业务流程是：用户向旅行社提出查询请求，旅行社向飞机场转发请求，然后飞机场将查询的结果返回给旅行，最后旅行社返回给用户。在“硬连线”的前提下，旅行社服务必须知道飞机场服务的具体信息。这时，如果飞机

场服务发生变化，比如飞机场服务不再直接向旅行社提供查询服务，而需要在中间插入一个售票点服务，那么旅行社服务的对象引用必须重新指向售票点服务，从而造成旅行社服务必须重新部署，无法完成系统的动态演化。相反，从体系结构的视角出发，旅行社服务应该引用的是能够完成飞机票查询功能的服务，而不管这个服务是飞机场服务还是后来的售票点服务。也就是说，这个对象引用的解释应该是根据体系结构的配置情况动态地加以改变。因此，要实现我们的体系结构对象与实际运行系统的因果互连，就要使体系结构对象介入基本服务之间的协同行为中，让它们之间的引用根据体系结构对象描述的上下文来灵活配置。

4 协同模型的自省式实现

4.1 体系结构对象的构造——元表示(meta-representation)

在我们的协同模型中，采用内置运行时体系结构对象来封装系统的自身描述，构成系统的元表示。基于此，在具体实现上，我们根据软件体系结构的组成并借鉴一些已有的软件体系结构语言(如 ACME)，来刻画、构成将软件体系结构的各个基本要素具体化(Reify)为面向对象程序设计语言中的类结构，从而使体系结构对象成为实际系统的元对象(meta-object)。

Component：代表了一个系统中的主要的计算单元或者是数据存储单元。在 SOC 的概念下，Component 指网络中通过各种方式构建的独立的软件服务(包括 EJB 构件、COM/COM+构件、CORBA 构件，也可以是使用任何方式实现的 Web Service)。Component 可能含有很多的接口，称为 Port，它们实际上定义了 Component 和环境的一个交互点。针对软件服务来说，即服务对外开放的接口。

Connector：用来表示 Component 之间的交互和协同活动。Connector 可以理解为负责软件服务之间消息传输的实体。Connector 中同样定义了一些接口，用来表示交互的参与者，我们称为 Role。在我们的协同模型中，主要包括了两种类型的 Role，称为 Caller 和 Callee，用来标记通信的方向。

Attachment：它并不是代表体系结构中的某个实体，而是封装了 Component 的 Port 和 Connector 的 Role 之间的匹配信息。

System：封装的是整个体系结构的配置信息(Configuration)。在我们的协同模型中，System 的概念通过类 RuntimeArchitecture 来描述，同时我们可以通过子类继承和接口实现的机制来引入特定的体系结构风格(Style)，具体见下文。所以，在对实际应用系统协同行为的刻画中，通常采用该类的一个具体子类的对象。

以上相关部分的类图如图 2 所示。

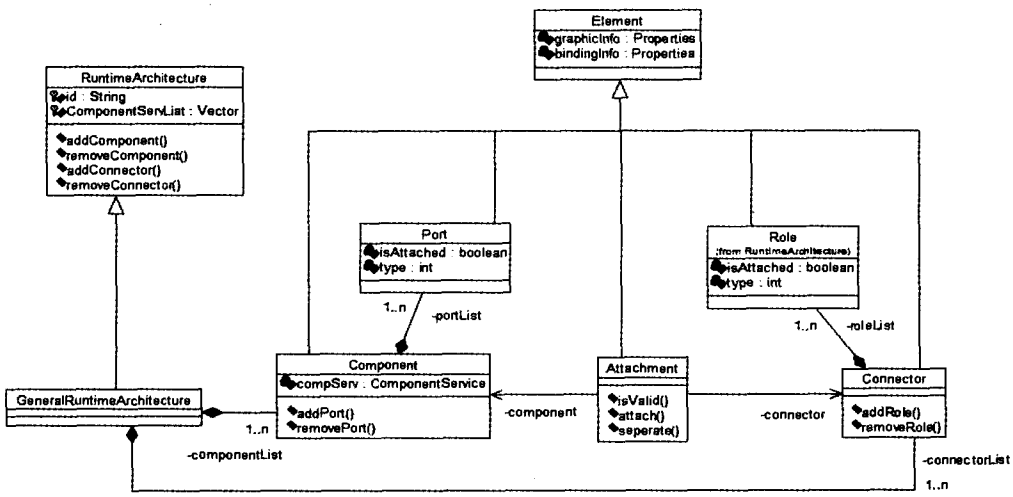

图 2 体系结构中基本元素的类图

4.2 体系结构对象的操纵——元协议(meta-protocol)

在我们的协同模型中，由于采用了体系结构类来刻画系统的元表示，因此作为对元表示进行操纵的元协议可以很自然地通过体系结构类中定义的方法来实现。但是，在自省的概念下，对元表示的操纵将最终反映到实际的运行系统中，所以我们期望对元协议的设计加入一定的约束，以避免在对元表示操纵前后体系结构的不一致性，最终破坏实际的运行系统。为此，我们加入了对体系结构风格的设计。体系结构风格是根据软件系统的结构组织定义的软件系统族，它通过组件应用的限制及其与组件有关的组成和设计规则来表现组件之间的关系，我们必须保证特定体系结构风格的应用系统在动态演化前后构件间相互关系的一致性得到满足，从而使元协议受到该风格的约束。

首先，我们为特定的体系结构风格增加软件服务组合时的组成元素方面的约束条件，并规定该约束条件应该在以后的系统运行中得到保持。在实现上，我们将满足特定的风格的体系结构对象设计为一般的体系结构对象的子类。比如，在简单的 Master - Slave 风格的体系结构中，我们认为，只能存在一个 Master，而 Slave 可以有多个，如下所示：

```
public class SimpleMasterSlaveType extends RuntimeArchitecture {
    Component master;
    Vector slaveList;
}
```

其次，我们通过在体系结构类上定义方法来实现对元协议的设计。值得注意的是，元协议的实现要满足前面指定的约束，不能破坏组件间的相互关系。比如，在简单的 Master - Slave 风格的体系结构中，Slave 的行为是改变相关联的 Master，而 Master 的行为是增加 Slave 和减少 Slave。我们把行为封装成相应的方法，并按组件的类型将方法组织成相应的接口。体系结构类实现这些接口，以加入特定的元协议。

```
public interface Slave {
```

```
    public Object changeMaster(Command command);
}
public interface Master {
    public void addSlave(Command command);
    public void removeSlave(Command command);
}
```

4.3 体系结构对象的实施——因果互连(causal connection)

正如上述,要在我们从体系结构角度构造的元层和构件所处的基层之间构造本质的因果互连关系,关键在于要使系统各个部件之间的引用放在体系结构对象所提供的上下文中解释。在实现层面上,表现为如何将我们的体系结构对象介入构件之间的交互之中。由此,我们希望在实际的运行系统中,各个部件之间的调用都通过内置的运行时体系结构对象来进行,避免系统构件间的"硬连线",即构件之间引用关系的解耦合。

基于以上的分析,我们采用 Java 语言中提供的 Dynamic Proxy[4]的机制,使对实际对象的方法调用都要间接地通过 Proxy 对象进行。而对于调用方来说,并不知道 Proxy 的存在,因为 Proxy 所展现的接口与请求接口完全吻合。具体来说,它以一种类型安全的方式去创建一个 Proxy,去代理一些指定接口的调用,并将这些调用分派到一个实现了 java.lang.reflect.InvocationHandler 接口的类,用 Reflection 的方式来完成实际的调用。

我们以通过 EJB 实现的软件服务之间的解耦合为例来介绍具体的实现机制。

第一阶段:部署阶段。对于没有请求其他 EJB 为其服务的 EJB,我们可以直接部署。然而对于有请求存在的 EJB,由于在它部署描述符中需要说明被请求的 EJB 的详细信息,因此不能直接部署。而我们要达到的目的就是避免直接引用被请求的 EJB。所以,我们在部署阶段先从提出请求的 EJB 中提取出被请求的 EJB 的 Home 接口,生成一个我们可以操纵的 Dynamic Proxy,完成对提出请求的 EJB 的部署。该过程的顺序如图 3 所示。

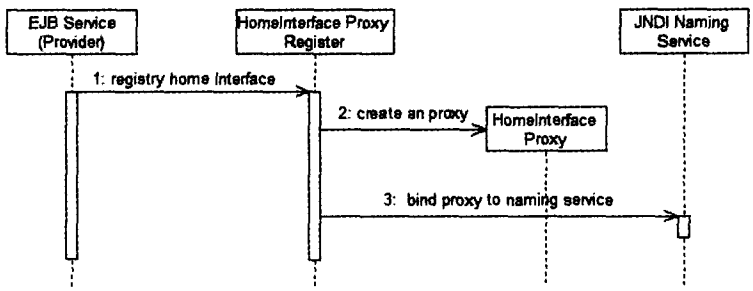

图 3 部署阶段:Home 接口的绑定过程

第二阶段:运行阶段。当我们将 Home 接口的 Proxy 绑定到 JNDI 上以后,请求方的 EJB 去 lookup 的时候,就会得到这个 Proxy 的引用。而当它调用 Home 接口的 create 方法的时候,我们定义的 HomeInterfaceProxyHandler 类就会捕获这个调用,创建被调用 EJB 的实例。但是并不将这个实例返回,而是保存在体系结构对象之中,并用这个实例 Remote 接口生成一个 Dynamic Proxy 返回,由体系结构对象去解释对这个 Remote 接口的调用。这样,体系结构对象将会捕获所有对 Remote 接口中方法的调用,从而实现了系统部件之间的解耦合。该过程的顺序图如图 4 所示。

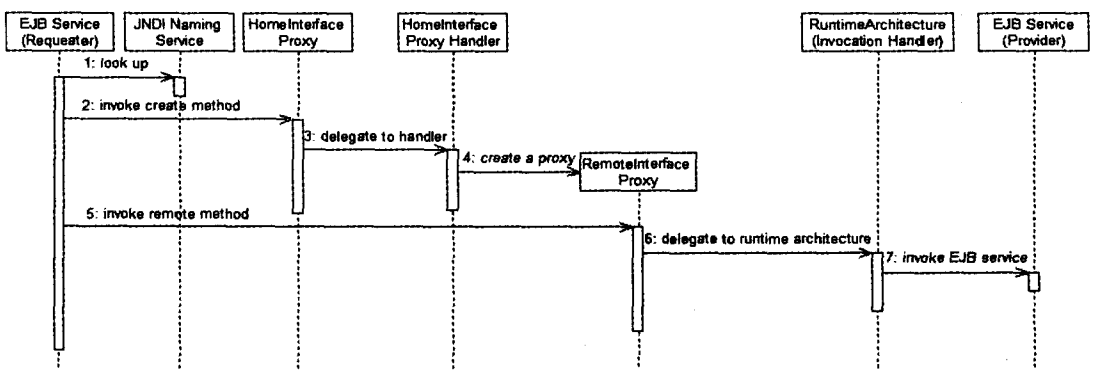

图 4 运行阶段:体系结构对象的插入过程

5 基于运行时体系结构的在线动态演化

采用运行时软件体系结构作为软件服务之间的协同模型以及基于计算自省的实现途径,目的是为面向服务的协同应用系统引入动态演化的能力。当我们将体系结构对象介入软件服务之间的协同行为之中,使系统各个部件之间的引用放在体系结构对象所提供的上下文中去解释,从而在体系结构和实际运行系统之间建立了本质的因果互连关系之后,实现系统的动态演化就显得比较自然了。我们可以将应用系统的动态演化行为理解为在体系结构提供的上下文中对部件之间的引用关系的重解释。

具体来说,当集成方需对系统进行重配置时,只需在集成端对软件体系结构对象进行演化即可。如上述的 Master/Slave 风格的体系结构类中,增加、去除 Slave 的演化行为直接定义为该类的方法 addSlave 和 removeSlave,这些方法将返回一个个具体的命令,然后将这些演化命令发送到各个主机的软件体系结构对象中。它们收到命令后,将在保证当前系统满足事务性要求的前提下对自己进行更新。这样的更新无

需重启系统，也不需要重新编译。以上的演化行为称为预设性的(planned)演化。相关过程的顺序图如图5所示。

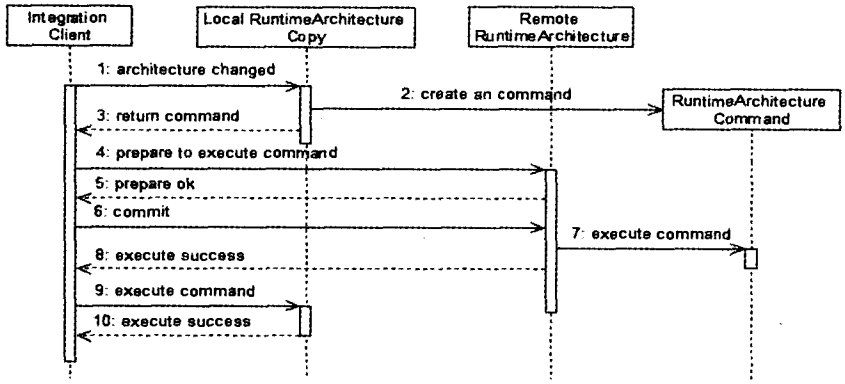

图5 基于运行时体系结构的预设性演化过程

此外，运行系统的演化需求并不总是能够在体系结构的设计阶段做出完整的预期。为此，我们也对非预设性的(unplanned)演化提供有效的支持，这主要是指当在特定的风格的体系结构对象中定义的演化动作不能满足用户的需求时，用户可以自己定制新的体系结构对象来反映新的演化动作，即非预设性的演化动作可以理解在用户自己定义的体系结构对象提供的上下文中对部件之间的引用关系进行重解释。

特别地，我们约定，这种定制只能使用该类型的子类来进行，以保证体系结构的兼容性。由于涉及到用户自定义的类，因此需要将该子类的class文件传到远程服务器上，否则服务器将无法识别用户自定义的类型。在远程服务器接收到class文件以后，集成方发出“升级”类型的命令。远程服务器在接收到该命令后，同样只能在系统允许更新的状态下进行“升级”。这样，原有的体系结构对象就可以具备新的子类型的所有演化行为的能力了。以后具体的演化过程与预设性演化相同。体系结构“升级”的相关过程的交互图如图6所示。

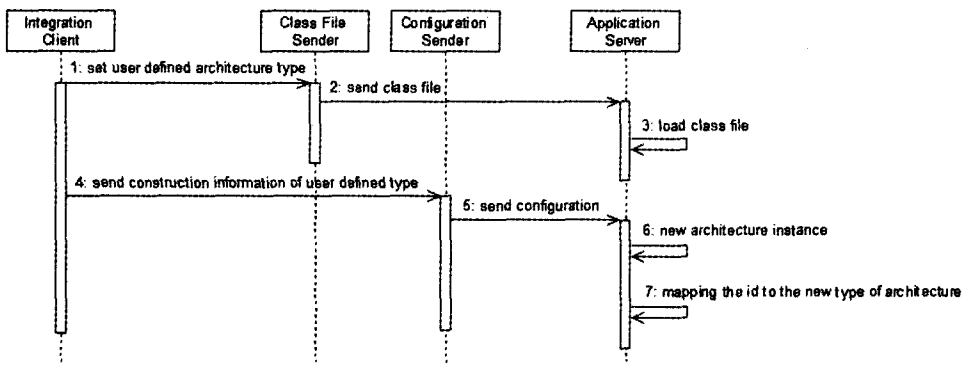

图6 非预设性演化的体系结构“升级”过程

结论及进一步的工作 Internet开放环境的特性给构造面向服务的计算提出了许多新的挑战，尤其是如何让面向服务的协同应用系统能够动态调整以适应环境和需求变化，更是其中的一个关键性问题。ARTEMIS-ARC系统基于运行时体系结构的协同模型以及相应的实现机制，给出了这个问题的一个较为令人满意的答案。但是，将体系结构从抽象的规约转化为切实的可操纵的对象实体，并确保对该对象实体操纵能确实反映到系统中，不是轻而易举的。

本文采用了自省的途径来解决上述问题。具体地说，我们通过将从体系结构角度观察的系统自描述信息封装成体系结构类，构建出系统的元表示，并结合体系结构风格在该类中定义相应的方法来实现元协议，完成对体系结构类的操纵。更进一步，借助Dynamic Proxy机制对基本服务之间的引用关系进行了解耦合，将体系结构对象介入到基本服务的协同之中，完成了体系结构对象和实际运行系统之间的因果互连关系的建立。在以上工作的基础上，本文还探讨了基于内置运行时体系结构的服务组合的动态演化，给出了较为一般性的动态演化过程描述。

在下一步的工作中，我们首先要完善体系结构的风格库，提供更多的体系结构风格选择。另外，我们希望能够进一步寻找基于工作流模型和基于体系结构模型之间的映射关系，希望能对面向工作流的系统提供动态演化的支持。

参考文献

1 Papazoglou M P, Georgakopoulos D. Service-oriented computing: Introduction. Communications of the ACM, 2003, 46(10): 24～28
2 南京大学软件研究所. 动态架构Web服务集成平台ARTEMIS-ARC设计说明书 v 1.1, Jan. 2005
3 Coulson G. What is reflective middleware? Distributed Systems Online Journal, 2000. Available at: http://boole.computer.org/dsonline/middleware/RMarticle1.htm
4 Blosser J. Explore the Dynamic Proxy API. Javaworld, Nov. 2000
5 David G, Robert T M, David W. Acme: Architectural Description of omponent-Based Systems. In: Foundations of Component-Based Systems. Cambridge University Press, 2000. 47～68
6 Ma X, Cao J, Chan A, et al. A Graph-Oriented Approach to the Description and Implementation of Distributed and Dynamic Software Architecture. In: The 15th International Conference on Software Engineering and Knowledge Engineering. San Francisco, USA, July 2003. 518～524
7 Garlan D, Cheng Shang-Wen, Huang An-Cheng, et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. IEEE Computer, 2004, 37(10): 46～54
8 马晓星，余萍，陶先平，等. 一种面向服务的动态协同架构及其支撑平台. 计算机学报，2005, 28(4): 467～477
9 Costa FM. Combining meta-information management and, reflection in an architecture for configurable and reconfigurable middleware: [Ph D Thesis]. Lancaster: Lancaster University, 2001