基于 Pi-演算的 BPEL4WS Web 服务组合形式化模型*)

辜希武　卢正鼎
(华中科技大学计算机科学技术学院　武汉 430074)

摘　要　Web服务组合研究领域的一个重要的问题是如何形式化描述Web服务组合，如何验证服务组合的正确性。Web服务组合的形式化模型来可以用来检查、验证Web服务组合以保证组合的正确性。Pi-演算是一种适合于Web服务组合建模的进程代数。本文介绍了Pi-演算的基本语法，针对目前最主要的一种描述和执行基于工作流模式的Web服务组合的规范-Web服务商业流程执行语言(Business Process Execution Language for Web Services，BPEL4WS)，定义了Pi-演算和BPEL4WS之间的概念映射，并给出了BPEL4WS的基于Pi-演算的形式化模型，最后通过一个案例给出了模型验证的方法。

关键词　Pi-演算，Web服务，Web服务组合，Web服务商业流程执行语言

A Pi-calculus Based Formal Model for BPEL4WS Web Service Composition

GU Xi-Wu　LU Zheng-Ding
(College of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074)

Abstract　One of important issue in Web service composition researching area is how to describe Web service composition formally and verify the correctness of Web service composition. A formal model of Web service composition can be used to check and verify Web service composition so that the correctness of Web service composition can be guaranteed. Pi-calculus is a kind of process algebra which is suitable for modeling Web service composition. This paper introduces basic syntax of Pi-calculus. For the most important kind of specification of specifying and executing workflow based Web service composition- Business Process Execution Language for Web Services(BPEL4WS), the paper defines the conception mapping between Pi-calculus and BPEL4WS and presents a Pi-calculus based formal model for BPEL4WS. The model verification method is also introduced through a case study.

Keywords　Pi-calculus, Web service, Web service composition, Business process execution language for Web services

1　介绍

面向服务的计算(Service Oriented Computing，SOC)[1]是一种新的分布式计算和电子商务模式。在这种计算模式里，自治、异构的服务是基本的计算单元，显然Web服务技术能够很好地适应和支持SOC。关于Web服务的研究包括很多令人感兴趣的问题，例如服务组合、服务同步、服务协作和服务验证。Web服务组合是指：一个客户或客户代理的请求不能由单个Web服务来满足，而是由一些Web服务的组合来完成。这些被组合的子Web服务会彼此并发地交互以完成客户的请求，而彼此间的交互是通过通信和交换信息来完成。因此Web服务组合涉及到的问题包括被组合的服务间的并发、同步、通信。

目前有很多研究者利用并发系统的形式化模型来建模、验证Web服务组合以保证Web服务组合的正确性。主要的模型有通信顺序进程[3](Communicating Sequential Processes，CSP)、通信系统演算[4](Calculus of Communicating Systems，CCS)、Pi-演算[5]、Petri网等。Web服务流语言[6](Web Services Flow Language，WSFL)是从工作流语言发展而来的基于网的语言，它可以被看成是着色Petri网。Hamadi和Benatallah[7]提出了一种基于Petri网的代数来组合Web服务。通过把每个服务组合操作映射到Petri网的构造元素，每个Web服务组合操作的形式化语义可以被Petri网来表达。因而，任何一个用基于Petri网的代数来表达的服务都可以被翻译成Petri网的表示形式。Brogi，Canal，Pimentel，等[8]提出了一个基于CCS的Web服务编排接口[9](Web Service Choreography Interface，WSCI)的形式化模型。Salaün，Bordeaux and Schaerf[10]提出了一个利用CCS对BPEL4WS建模和推理的通用框架。Cámara，Canal和Cubo[11]利用CCS对BPEL4WS建模，并且利用形式化模型来分析Web服务行为的兼容性和可替换性。Foster，Uchitel，Magee，等[12]将BPEL4WS所描述的商业流程翻译成FSP(Finite State Process)，并且利用LTSA(Labeled Transition System Analyzer)工具对FSP进程进行模型检查。

Pi-演算是Robin Milner提出的一种用于描述并发进程间的可移动通信的进程代数。它和传统的进程代数如CSP、CCS一样，是用来描述并发的进程间的互相交互(通信)的数学模型。但它和传统进程代数的一个重要区别就是它对并发进程间的通信的可移动性(Mobility)的描述能力。在传统的进程代数里，进程间的通信是在某个固定不变的通道(Chan-

*)基金资助：国家自然科学基金资助项目(项目编号：60403027)，湖北省自然科学基金(项目编号：2005ABA258)，软件工程重点实验室开放基金(项目编号：SKLSE05-07)。辜希武　博士研究生，研究方向为Web服务、语义Web和中间件；卢正鼎　教授，博士生导师，CCF会员，研究方向为分布式系统集成，Web数据管理，信息安全，数据库系统。

nel)上进行的,因此传统的进程代数描述的是一个静态的拓扑结构,在这个静态的拓扑结构里所有进程间的交互是在固定不变的通道上进行的。而由 CCS 发展而来的 Pi-演算则把通道名看作是普通的变量和值而不加区分,因此通道名和普通的变量一样可以在进程之间传递,而进程间可以在新传递的通道上进行交互通信。因此,Pi-演算描述的是一个动态的拓扑结构,进程间通信的通道是可以随时变化和迁移的。这使得 Pi-演算特别适合于描述那些结构不断变化的并发系统。

作为一个事实上的工业标准,BPEL4WS[2]成为了基于工作流模式的 Web 服务组合的重要规范。从技术上讲,BPEL4WS 缺乏一个形式化的模型来对组合的 Web 服务进行模型检查和验证。因此,本文提出了一个基于 Pi-演算的 BPEL4WS 的形式化模型,在该模型中,对 BPEL4WS 中用于描述一个商业流程的基本构造元素—行为(activity)进行了形式化描述。本文还通过一个案例给出了模型验证方法。

本文包括以下内容:第 2 节介绍了 Pi-演算的基本语法;第 3 节定义了 Pi-演算到 BPEL4WS 的概念映射,给出了基于 Pi-演算的 BPEL4WS 的形式化描述;第 4 节通过案例分析给出了对 Web 服务组合模型进行验证的方法;最后给出全文总结和未来的工作。

2 Pi-演算的基本语法

Pi-演算理论的主要内容包括:语法定义、结构等价规则、操作语义、互相似理论、代数理论等。下面我们给出它的基本语法定义。Pi-演算理论的其它内容由于篇幅所限不在此详细介绍。

在 Pi-演算里,基本的计算实体为名字(变量)和进程。设 N 为一个无限的名字集,在 N 上的小写字母如 $a,b,\cdots,z$ 代表所有的通信通道、变量和值,大写字母如 $P,Q,\cdots$ 代表进程或进程表达式。Pi-演算的语法定义如下:

$$P::=0\mid\bar{a}x.P\mid a(x).P\mid\sum_{i\in I}P_i\mid[x=y]P\mid[x\neq y]P\mid(\nu x)P\mid !P\mid A(y_1,\cdots,y_n)$$

其中:

1. 在 Pi 演算里定义了三种前缀动作。输出前缀 $\bar{a}x$ 表示名字 x 沿着通道 a 输出;输入前缀 $a(x)$ 表示从通道 a 接受一个名字,该名字将替换名字 x。前缀 τ 表示一个进程外部不可见的内部动作(或称为哑动作)。

2. 空进程 0 表示不执行任何动作。

3. $\alpha.P$ 表示先执行前缀动作 α,然后执行进程 P。在 Pi-演算里,符号. 用来表示顺序执行。

4. $\sum_{i\in I}P_i$ 表示选择执行,I 为有限索引集合。例如 P_1+P_2 表示选择执行 P_1 或 P_2。

5. 匹配表达式 $[x=y]P$ 表示一个进程,当名字 x 和 y 为同一个名字时,其行为表现为进程 P。匹配表达式 $[x\neq y]P$ 表示一个进程,当名字 x 和 y 不为同一个名字时,其行为表现为进程 P。

6. 约束 $(\nu x)P$ 表示名字 x 被局限在进程 P 内部。特别地,如果 x 代表一个通道名,则该通道只能在 P 的内部通信中使用。在 P 的外部,通道 x 是不可见的。在 P 的内部通道 x 上进行的通信对 P 的外部而言全部是哑动作 τ。

7. $!P$ 表示 P 被重复复制无穷多次。

8. $A(y_1,\cdots,y_n)$ 称为进程标识符。每个进程标识符有其定义。假如进程标识符 $A(y_1,\cdots,y_n)$ 的定义为 $A(x_1,\cdots,x_n)::=P(i\neq j=>x_i\neq x_j)$。其含义为:当用 y_i 替换 x_i 时,$A(y_1,\cdots,y_n)$ 的行为将表现为进程 P。因此进程的定义就像是进程的声明($x_1,\cdots,x_n$ 为形参),进程标识符就像是带实参 $y_1,\cdots,y_n$ 的进程调用。

Pi-演算将名字分为二类:绑定名(bound name)和自由名(free name)。被约束的名字和输入动作的名字称为绑定名,其余的名字称为自由名。对进程 P,其绑定名的集合记为 $bn(P)$,其自由名的集合记为 $fn(P)$。进程标识符和进程定义中带的参数名只能为自由名。

Pi-演算将名字替换记为 $\{\vec{y}/\vec{x}\}$。其中 $\vec{y}$ 表示 $(y_1,\cdots,y_n)$,$\vec{x}$ 表示 $(x_1,\cdots,x_n)$。$P\{\vec{y}/\vec{x}\}$ 表示对进程 P 进行变量替换,$\vec{y}$ 中的 y_i 替换 $\vec{x}$ 中的 x_i。

在 Pi-演算理论中,$P\xrightarrow{\alpha}Q$ 表示进程 P 经过动作 α 变迁(演化)成为进程 Q,其中动作 α 可以为内部哑动作(τ)、输入和输出。特别地:输入动作 $a(x)$ 和输出动作 $\bar{a}u$ 正好是一对互补动作,输出名 u 沿着通道 a 正好被输入动作 $a(x)$ 输入以替换名字 x,导致进程的名字替换 $\{u/x\}$,同时这对互补的动作互相抵消而成为一个内部动作 τ。在 Pi-演算理论里这种情况可以被表示为:

$$\frac{P\xrightarrow{a(x)}P',Q\xrightarrow{\bar{a}u}Q'}{P\mid Q\xrightarrow{\tau}P'\{u/x\}\mid Q'}$$

其中分子部分为前提,分母部分为结论。

3 基于 Pi-演算的 BPEL4WS 的形式化描述

BPEL4WS 作为一种描述和执行 Web 服务组合工作流的语言,正在成为事实上的标准。BPEL4WS 可以看作一种编制(orchestration)语言,为了将业务逻辑相关的子服务组合成一个新的 Web 服务,服务的开发者可以利用 BPEL4WS 规范定义一个新的服务,该服务可以被称为编制者(orchestrator),新的服务处于一个中心协调者的地位,它负责和各个子服务交换信息,协调这些子服务间的交互以完成组合服务的功能。BPEL4WS 定义了各种不同的原子行为(atom activities)、结构化的行为(structural activities)和并发行为(flow activities)来描述一个组合的商业流程。同时它还提供了异常处理器(faulthandler)、事件处理器(eventhandler)和补偿(compensationhandler)来处理异常、事件和事务。

在用 Pi-演算对 BPEL4WS 中定义的行为和处理器进行形式化建模前,首先建立一个 BPEL4WS 的概念和 Pi-演算的概念之间的映射关系如下:

1. BPEL4WS 描述的一个商业流程(process)和一个 Pi-演算进程相对应。

2. BPEL4WS 规范中定义的各种行为(activities)和一个 Pi-演算进程相对应。

3. BPEL4WS 规范中定义的各种处理器(handler)和 Pi-演算进程相对应。

4. 用 Pi-演算进程间通信来描述 Web 服务间的交互关系。

5. BPEL4WS 中的定义的变量(variables)可以被映射为 Pi-演算进程间通信所交换的信息。

6. BPEL4WS 中的 partnerLink、portType 和 operation 描述了互相交互的 Web 服务的访问点,它们和 Pi-演算进程通信使用的通道相对应。

为了方便对 BPEL4WS 进行形式化描述,我们首先给出下面两个定义:

定义 1 对于一个三元组 $AccessPoint=(partnerLink, portType, operation)$，函数 $Chan(AccessPoint)$ 返回一个唯一的通道名。

定义 2 函数 $|\rangle_{Pi}$ 为一个从 BPEL4WS 的行为到一个 Pi-演算进程的映射，即对于一个 BPEL4WS 描述的一个行为 A_{bpel}，$|A_{bpel}\rangle_{Pi}$ 是该行为对应的 Pi-演算进程。

下面我们对 BPEL4WS 中定义的各种行为给出基于 Pi-演算的形式化模型。为了便于表达，在给出每个行为的 BPEL4WS 定义时，我们只列出一些主要属性和子元素，其他属性和子元素用省略号省略。

3.1 原子行为

3.1.1 *Invoke*

Invoke 行为将调用一个 Web 服务，其定义如下：

```
⟨invoke
    partnerLink="ncname"
    portType="qname"
    operation="ncname"
    inputVariable="ncname"
    outputVariable="ncname"…⟩
/invoke⟩
```

Invoke 行为里的 *partnerLink*、*portType*、*operation* 指定了要调用服务的访问点，*inputVariable* 指定了调用输入参数，*outputVariable* 返回调用结果。我们可以把该行为建模为：调用者通过通道 $chs=Chan(partnerLink, portType, operation)$ 发送输入变量和一个被约束的通道名 *ret*，然后从通道 *ret* 输入调用结果。用一个 Pi-演算进程表示为：

$$|Invoke\rangle_{Pi}=(\nu ret)(\overline{chs}\langle inputVariable.\overline{chs}\langle ret\rangle\rangle \mid ret(outputVariable))$$

3.1.2 *Receive*

Receive 行为将等待一个调用请求，其定义如下：

```
⟨receive
    partnerLink="ncname"
    portType="qname"
    operation="ncname"
    variable="ncname"…⟩
/receive⟩
```

Receive 行为里的 *partnerLink*、*portType*、*operation* 指定了要调用服务的访问点，*variable* 用于接受请求参数。我们可以把该行为建模为：等待请求者通过 $chs=Chan(partnerLink, portType, operation)$ 接受请求参数和和一个通道名 *ret* 以用于将来返回结果(如果需要)。用一个 Pi-演算进程表示为：

$$|Receive\rangle_{Pi}=chs(variable).chs(ret)$$

3.1.3 *Reply*

Reply 行为将返回一个应答，其定义如下：

```
⟨reply
    partnerLink="ncname"
    portType="qname"
    operation="ncname"
    variable="ncname …⟩
/reply⟩
```

Reply 行为里的 *partnerLink*、*portType*、*operation* 指定了要调用服务的访问点，*variable* 指定了返回的应答结果，我们可以把该行为建模为：应答者通过通道 $chs=Chan(partnerLink, portType, operation)$ 返回应答结果。用一个 Pi-演算进程表示为：

$$|Reply\rangle_{Pi}=\overline{chs}\,variable$$

3.1.4 *Assign*

Assign 行为用于变量的赋值，其定义如下：

```
⟨assign …⟩
    ⟨copy⟩
        ⟨from variable="ncname1"⟩
        ⟨to variable="ncname2"⟩
    ⟨/copy⟩
⟨/assign⟩
```

我们可以用 Pi-演算里的一对互补的输入输出动作来模拟变量间的赋值，建立一个内部通道 *assign*，将变量 *ncname1* 作为通道 *assign* 上的输出，变量 *ncname2* 作为通道 *assign* 上的输入。用一个 Pi-演算进程表示为：

$$|Assign\rangle_{Pi}=(\nu assign)(\overline{assign}\,ncname1 \mid assign(ncname2))$$

3.1.5 *Throw*

Throw 行为用于抛出一个异常，其定义如下：

```
⟨throw
    faultName="qname"
    faultVariable="ncname" …⟩
    …
⟨/throw⟩
```

在 BPEL4WS 的异常处理机制里，每个异常都有一个唯一的异常名字 faultName，并且带有相关的异常参数 *faultVariable*。异常处理器将根据异常名来捕获异常。为了模拟异常抛出和捕获机制，我们建立通道 *faulName*(注意该通道名是唯一的)，当有异常需要抛出时，则从通道 *faulName* 输出异常变量。相应的异常处理器则监听通道 *faulName*，如果有输入就执行相应的异常处理行为。用一个 Pi-演算进程建模 *Throw* 行为如下：

$$|Throw\rangle_{Pi}=\overline{faultName}\,faultVariable$$

3.1.6 *Empty*

Empty 行为代表一个没有任何动作的行为，因此：$|Empty\rangle_{Pi}=0$

3.1.7 *Compensate*

Compensate 行为用于调用一个 *CompensationHandler*，其定义如下：

⟨*compensate scope*="*ncname*" …/⟩

其中 *scope* 名是唯一的，所以我们以 *scope* 名建立通道，在通道上输出一个信号(以⟨⟩表示)来模拟调用行为。设 *scope* 名为 S，因此用一个 Pi-演算进程建模 *Compensate* 行为如下：

$$|Comensate\rangle_{Pi}=\overline{S}\langle\rangle$$

3.2 结构化行为

3.2.1 *Sequence*

Sequence 定义了一组顺序执行的子行为，其定义如下：

```
⟨sequence …⟩
    …
    activity+
⟨/sequence⟩
```

其中 $activity^+$ 表示一个或多个子行为。我们用 $Sequence(A_1,\cdots,A_n)(n\in I, I$ 为自然索引集$)$ 表示由 n 个 BPEL 子行为 $A_1,\cdots,A_n$ 组成的一个顺序执行的序列，则用 Pi-演算建模如下：

$$|Sequence(A_1,\cdots,A_n)\rangle_{Pi}=|A_1\rangle_{Pi}\cdot\cdots\cdot|A_n\rangle_{Pi}$$

3.2.2 *Switch*

Switch 行为根据一个或多个选择条件分支，从若干个子行为中选择一个执行。其定义为：

```
⟨switch …⟩
    …
    ⟨case condition="bool-expr"⟩+
        activity
    ⟨/case⟩
```

```
    〈otherwise〉 activity 〈/otherwise〉
〈/switch〉
```

Switch 里的条件判断有 1 个或多个，条件值为布尔表达式。在 Pi-演算里我们可以用名字比较来模拟条件表达式。设 n 为条件的个数，$S_N=\{N_1,\cdots,N_n\}$为 n 个条件值的集合，$S_A=\{A_1,\cdots,A_n\}$为相应条件条件满足时要执行的 BPEL 子行为的集合，D 为条件全部不满足时要执行的缺省的 BPEL 子行为，则 BPEL 的 *Switch* 行为可以表示为 $Switch(S_N, S_A, D, n)$，则用 Pi-演算建模如下：

$$|Switch(S_N,S_A,D,n)\rangle_{Pi}=\sum_{i=1}^{n}[x=N_i]|A_i\rangle_{Pi}+[x\neq N_1]\cdots[x\neq N_i]|D\rangle_{Pi}$$

3.2.3 *While*

While 行为允许其包含的子行为被重复执行，直到某个条件不再满足。其定义如下：

```
〈while condition="bool-expr"…〉
    …
    activity
〈/while〉
```

设名字 N 为迭代终止条件，*While* 包含的 BPEL 子行为为 A，我们用 $While(N, A)$来表示 *While* 行为。为了建模迭代行为，必须用 *Pi*-演算进程的递归定义，所以 $While(N, A)$ 可以被建模为：

$$|While(N,A)\rangle_{Pi}=[x=N]|A\rangle_{Pi}.\ |While(N,A)\rangle_{Pi}+[x\neq N]0$$

3.2.4 *Pick*

Pick 行为等待一个事件集合中的一个事件的发生，然后完成和发生的事件相联系的行为。每个 *pick* 元素至少包括一个 *onMessage* 元素，*Pick* 行为定义如下：

```
〈pick …〉
    〈onMessage
        partnerLink="ncname"
        portType="qname"
        operation="ncname"
        variable="ncname"?〉+
            activity
    〈/onMessage〉
〈/pick〉
```

定义中的上标＋表示可以有一个或多个事件发生。*onMessage* 元素的语义和 *receive* 的行为的语义完全一样，可以被建模为 Pi-演算的输入事件。*partnerLink*、*portType*、*operation* 指定了事件接受的通道，*variable* 指定了事件参数，*activity* 指定了事件发生后执行的行为。接受事件的通道 $che=Chan(partnerLink, portType, operation)$。设三元组集合 $S_E=\{(che_i, var_i, A_i)\mid i\in I, I$ 为自然索引$\}$，其中 che_i 为第 i 个事件对应的通道，var_i 为第 i 个事件对应的事件变量，A_i 为第 i 个事件对应的 BPEL 行为。设事件个数为 n，则 Pick 行为可以表示为 $Pick(S_E, n)$，$Pick(S_E, n)$可以被建模为：

$$|Pick(S_E,n)\rangle_{Pi}=\sum_{i=1}^{n}che_i(var_i).\ |A_i\rangle_{Pi}$$

3.3 并发行为 *Flow*

Flow 定义了一组并发执行的子行为，其定义如下：

$$\langle flow\cdots\rangle\cdots\ activity^{+}\langle/flow\rangle$$

其中 $activity^{+}$ 表示一个或多个子行为。我们用 $Flow(A_1, \cdots, A_n)(n\in I, I$ 为自然索引集$)$表示由 n 个 BPEL 并发子行为 $A_1, \cdots, A_n$ 组成的行为，则用 Pi-演算建模如下：

$$|Flow(A_1,\cdots,A_n)\rangle_{Pi}=|A_1\rangle_{Pi}|\cdots||A_n\rangle_{Pi}$$

3.4 异常处理 *FaultHandler*

FaultHandler 的定义如下：

```
〈faultHandlers〉
    〈catch
        faultName="qname"
        faultVariable="ncname"〉+
            activity
    〈/catch〉…
〈/faultHandlers〉
```

定义中的上标＋表示有一个或多个异常。当 *FaultHandler* 捕获异常后，会根据 *faultName*(唯一的)执行相应的行为，*faultVariable* 指定了相应错误信息。设三元组集合 $S_F=\{(faultName_i, var_i, A_i)\mid i\in I, I$ 为自然索引$\}$，其中 $faultName_i$ 为第 i 个错误名(也就是监听异常的通道名)，var_i 为第 i 个异常对应的错误信息，A_i 为第 i 个错误名对应的 BPEL 行为。设错误名个数为 n，则 *FaultHandler* 行为可以表示为 $FaultHandler(S_F, n)$，$FaultHandler(S_F, n)$可以被建模为：

$$|FaultHandler(S_F,n)\rangle_{Pi}=\sum_{i=1}^{n}faultName_i(var_i)\cdot|A_i\rangle_{Pi}$$

3.5 事件处理 *EventHandler*

EventHandler 的定义为：

```
〈eventHandlers〉
    〈onMessage
        partnerLink="ncname"
        portType="qname"
        operation="ncname"
        variable="ncname"?〉+
            activity
    〈/onMessage〉…
〈/eventHandlers〉
```

定义中的上标＋表示可能有一个或多个事件发生。*onMessage* 元素的语义和 Pick 行为里的 *onMessage* 元素语义完全一样。因此，*partnerLink*、*portType*、*operation* 指定了事件接受的通道，*variable* 指定了事件参数，*activity* 指定了事件发生后执行的行为。接受事件的通道 $che=Chan(partnerLink, portType, operation)$。设三元组集合 $S_E=\{(che_i, var_i, A_i)\mid i\in I, I$ 为自然索引$\}$，其中 che_i 为第 i 个事件对应的通道，var_i 为第 i 个事件对应的事件变量，A_i 为第 i 个事件对应的 BPEL 行为。设事件个数为 n，则 *EventHandler* 行为可以表示为 $EventHandler(S_E, n)$，$EventHandler(S_E, n)$可以被建模为：

$$|EventHandler(S_E,n)\rangle_{Pi}=\sum_{i=1}^{n}!\ che_i(var_i).\ |A_i\rangle_{Pi}$$

要注意的是 *EventHandler* 应该是不停的监听事件的发生并执行相应的行为，因此和 Pick 行为不同的是 $che_i(var_i).\ |A_i\rangle_{Pi}$ 是被不断复制的。

3.6 补偿处理 *CompensationHandler*

CompensationHandler 定义如下：

```
〈compensationHandler〉
    activity
〈/compensationHandler〉
```

一个 CompensationHandler 是和一个唯一的 scopename 相联系。CompensationHandler 是通过 compensate 来调用的，因此 CompensationHandler 必须监听通道 scopename 上的信号，到收到信号再执行相应的行为。设 CompensationHandler 要执行的行为为 A，对应的 scopename 为 S，则 CompensationHandler(S, A)被建模为：

$$|Compensation(S,A)\rangle_{Pi}=S(\langle\rangle).\ |A\rangle_{Pi}$$

4 Web服务组合模型的验证

Web服务组合模型的验证是为了保证那些并发的子服务相互之间交互的正确性。验证的内容主要包括：

模型的正确性：如同步是否正确？是否存在死锁？消息是否会丢失？

行为的等价性：两个Web服务行为等价的是指从Web服务的使用者的角度来看这两个Web服务的行为完全一致。在进程代数中，有两种行为等价性的定义：强相似(strong bisimilarity)和弱相似(weak bisimilarity)。两个进程 P 和 Q 强相似是指：P 能完成的所有动作和变迁(内部的和外部的)Q 都能完成。两个进程 P 和 Q 弱相似是指：外部观察者所能观察到的 P 和 Q 的外部行为和变迁是一致的。Pi-演算以形式化的方法给出了两个进程等价的精确定义[14]。服务行为等价性判定的意义在于自动搜索、匹配和重新组合新的等价的Web服务。

下面我们以一个旅行社代理服务为例来给出用Pi演算对Web服务组合的建模以及验证的方法。

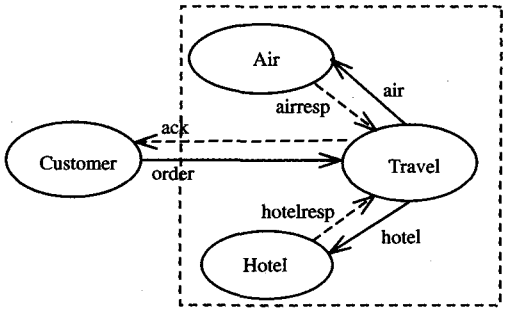

图1 用Pi-演算流图描述的Web服务组合

一个典型的旅行社代理Web服务组合业务流程如下：客户通过通道 *order* 发出一个订单请求，旅行社收到订单后分别向航空公司和酒店订机票和房间。航空公司和酒店会将处理结果将返回给旅行社。最后旅行社返回结果给客户。我们用Pi-演算流图来描述这个组合的业务流程如图1所示，图中椭圆代表进程，实线为进程间固定通信通道，虚线为动态通道。从客户的角度来看，与之交户的服务是一个单一的组合的服务，组合服务和客户的间交互的通道为 *order* 和 *ack*。*Travel* 服务作为一个中心的协调者来负责调用和组合子服务 *Air* 和 *Hotel*，其内部通道 *air*、*hotel*、*airresp*、*hotelresp* 以及内部交互对客户而言不可见。用BPEL4WS描述的 *Travel* 服务如下所示：

```
〈process name="Travel" …〉
  〈sequence〉
    〈receive
       partnerLink="order"
       variable="ack" …/〉
    〈receive
       partnerLink="order"
       variable="orderreq" …/〉
    〈flow〉
       〈sequence〉
         〈invoke
           partnerLink="air"
           outputVariable="resulta"
           inputVariable="orderreq"…/〉
         〈reply
           partnerLink="order"
           variable="resulta" …/〉
       〈/sequence〉
       〈sequence〉
         〈invoke
           partnerLink="hotel"
           outputVariable="resulth"
           inputVariable="orderreq"…/〉
         〈reply
           partnerLink="order"
           variable="resulth" …/〉
       〈/sequence〉
    〈/flow〉
  〈sequence/〉
〈/process〉
```

为了和下面要用到的模型验证工具的语法规则保持一致，我们用 $'a\langle x\rangle$ 表示从 a 通道输出名字 x，用 ν 表示一个约束名(内部名)。

1. *Customer* 沿着通 *order* 向 *Travel* 发送订单请求 *orderreq* 和动态通道名 *ack*，同时在通道 *ack* 等待结果。用一个Pi-演算的进程来建模 *Customer* 如下：

$$Customer(order,orderreq,ack)=$$
$$'order\langle ack\rangle.'order\langle orderreq\rangle.$$
$$(ack(resulta).0\,|\,ack(resulth).0)$$

2. *Travel* 从通道 *order* 收到通道名 *ack* 和订单请求 *orderreq* 后，分别沿着通道 *air*、*hotel* 向 *Airline* 和 *Hotel* 发送订单和内部通道 *airresp*、*hotelresp*，并从内部通道 *airresp*、*hotelresp* 等待处理结果 *resulta* 和 *resulth*。最后把结果返回 *Customer*。用一个Pi-演算的进程来建模 *Travel* 如下：

$$Travel(order,air,hotel,ack)=$$
$$(\nu airresp,hotelresp)order(ack).order(orderreq).('air\langle orderreq\rangle.'air\langle airresp\rangle.airresp(resulta).'ack\langle resulta\rangle.0\,|\,'hotel\langle orderreq\rangle.'hotel\langle hotelresp\rangle.hotelresp(resulth).'ack\langle resulth\rangle.0)$$

3. *Airline* 从通道 *air* 收到订单 *orderreq* 和内部通道名 *airresp* 后，根据票务情况沿通道 *airresp* 向 *Travel* 返回结果。用一个Pi-演算的进程来建模 *Airline* 如下：

$$Airline(air,resulta)=$$
$$(\nu airresp)air(orderreq).air(airresp).$$
$$'airresp\langle resulta\rangle.0$$

同样地，用一个Pi-演算的进程来建模 *Hotel* 如下

$$Hotel(hotel,resulth)=$$
$$(\nu hotelresp)hotel(orderreq).$$
$$hotel(hotelresp).hotelresp\langle resulth\rangle.0$$

4. 由 *Travel*、*Airline* 和 *Hotel* 所组成的组合服务 *TravelService* 表示如下：

$$TravelService\ (order,resulta,resulth)=$$
$$Travel\mid Airline\mid Hotel=$$
$$(\nu air,hotel,airresp,hotelresp)(order(ack).order(orderreq).('air\langle orderreq\rangle.'air\langle airresp\rangle.airresp(resulta).'ack\langle resulta\rangle.0\,|\,'hotel\langle orderreq\rangle.'hotel\langle hotelresp\rangle.hotelresp(resulth).'ack\langle resulth\rangle.0)\mid air(orderreq).air(airresp).'airresp\langle resulta\rangle.0\mid hotel(orderreq).hotel(hotelresp).'hotelresp\langle resulth\rangle.0)$$

对Web服务形式化建模是为了验证服务的正确性。模型验证可以用在两个方面：一方面在抽象设计阶段模型可以清晰地描述服务的交互行为，模型验证可以保证设计阶段对服务交互行为描述的正确性；另一方面，当服务被具体实现后，我们可以从服务的具体实现中抽取出抽象模型，并和设计时模型进行等价性验证，以检查具体实现的正确性。

和其他强大而成熟的形式化模型一样，Pi-演算有着许多支持自动模型检查和验证的工具。Mobility Workbench[13](MWB)就是一个用来分析基于Pi-演算描述的并发系统的工具。利用该工具我们可以对一个基于Pi-演算描述的Web服

务组合进行验证以排查出可能存在的错误，可以跟踪服务的行为以及验证服务的等价性。MWB采用标准元语言(Standard ML)开发，运行在新泽西标准元语言编译器(New Jersey SML Compiler version 11.0)环境下。我们利用该工具验证的环境为Windows 2000 Server，采用MWB'99 4.137版。

利用MWB，我们首先可以对Web服务的模型进行语法检查以排除不正确的Pi-演算表达式。然后可以利用*deadlocks*命令检查组合的*Web*服务是否存在死锁，用*step*命令模拟、跟踪服务的交互行为。经过*deadlocks*命令检查，我们所描述的*Travel*、*Hotel*、*Air*、*TravelService*、*Custom*、以及整个闭合系统*TravelService* | *Custom*都不存在死锁。

模型检查另一个重要内容是验证所设计的组合服务是否确实符合客户需求。因此我们需要检查组合的*TravelService*服务和客户进程*Custom*之间能否正确的交互，即验证组合服务*TravelService*和*Custom*之间行为的匹配性，。一种直接的办法是用step命令对闭合系统*TravelService* | *Custom*进行跟踪，观测每一步可能的交互。但对于一个复杂的系统这种办法显然不可取。文献[10]提出了一个简单有效的方法：将Web服务的客户进程所有行为反转(输出变输入，输入变输出)，这样就得到客户进程*Custom*的镜像进程*RevCustom*。如果*RevCustom*进程和*TravelService*服务是弱相似的，那么*TravelService*和*Custom*就是匹配的。

我们首先将*Custom*进程反转，得到其镜像进程*RevCustom*如下：

$RevCustom\ (order, resulta, result) =$

$order(ack).order(orderreq).('ack\langle resulta\rangle.0 \mid 'ack\langle result\rangle.0)$

对比*TravelService*，我们可以看到*TravelService*中的内部动作$'air\langle orderreq\rangle$从内部通道*air*输出名字*orderreq*，同时内部动作$air(orderreq)$从内部通道*air*输入名字*orderreq*，因此这两个动作互相抵消而成为对外不可见的哑动作τ；同理，$'air\langle airresp\rangle$和$air(airresp)$、$'airresp\langle resulta\rangle$和$airresp(resulta)$、$'hotel\langle orderreq\rangle$和$hotel(orderreq)$、$'hotel\langle hotelresp\rangle$和$hotel(hotelresp)$、$hotelresp(result)$和$'hotelresp\langle result\rangle$全部被互相抵消而成为哑动作。其进程变迁可以表示为

$$TravelService \xrightarrow{\tau^+}$$

$order(ack).order(orderreq).$

$('ack\langle resulta\rangle.0 \mid 'ack\langle result\rangle.0) \mid 0 \mid 0$

其中τ^+代表一个或多个τ动作。因此，根据弱相似的定义*RevCustom*和*TravelService*是弱相似的。在MWB运行环境下通过运行weq TravelService RevCustom命令，也验证了*RevCustom*和*TravelService*是弱相似的。因而*TravelService*和*Custom*之间是匹配的。

总结和将来的工作 一个组合的Web服务由一组子Web服务组成。这些被组合的子Web服务会彼此并发地交互以完成客户的请求，而彼此间的交互是通过通信和交换信息来完成。Web服务组合一个很重要的问题是如何保证组合的正确性。解决这个问题的重要手段就是为Web服务组合提供一个形式化的模型以进行模型检查和验证。Pi-演算是一种特别适合于描述具有移动性的并发系统的进程代数，因而可以用来作为描述Web服务组合的形式化工具。本文利用Pi-演算对Web服务组合建立形式化模型，并且对目前最为重要的Web服务组合规范BPEL4WS，定义了Pi-演算到BPEL4WS的概念映射，给出了基于Pi-演算的形式化描述。并且通过案例分析给出了模型的验证方法。

将来的研究工作包括：如何处理Web服务组合中的时间因素和事务；如何利用Pi-演算建立Web服务自动组合的模型以及各种Web服务组合描述语言和Pi-演算模型的自动转换等问题。

参考文献

1 Huhns M N, Singh M P. Service-Oriented Computing: Key Concepts and Principles. ROBERT E. FILMAN. IEEE Internet Computing, 2005,9:75～81

2 Andrews T, Cubera F, Dholakkia H, et al. Business Process Execution Language for Web Services Version1.1. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, 2003-05-05.

3 Hoare C A R. Communicating Sequential Processes. New York: Prentice Hall, 2004

4 Nilner R. Communication and Concurrency. New York: Prentice Hall, 1989

5 Milner R. Communicating and Mobile Systems: The Pi Calculus. Cambridge: Cambridge University Press, 1999

6 Leymann F. Web Services Flow Language. http://www-306.ibm.com/software/solutions/Webservices/pdf/WSFL.pdf, 2005-10-20

7 Hamadl R, Benatallah B. A Petri Net-based Model for Web Service Composition. In: Proceedings of Fourteenth Australasian Database Conference on Database Technologies, Australia, 2003

8 Brogi A, Canal C, Pimentel E, et al. Formalizing Web Service Choreographies. CAVALCANTI ANA, MACHADO PATRICIA. Electronic Notes in Theoretical Computer Science, 2004, 105:73～94

9 Arkin A, Askary S, Fordin S, et al. Web Service Choreography Interface (WSCI)1.0. http://www.w3.org/TR/wsci/, 2006-3-10

10 Salaün G, Bordeaux L, Schaerf M. Describing and Reasoning on Web Services using Process Algebra. In: Proceedings of the IEEE International Conference on Web Services (ICWS'04), USA, 2004

11 Cámara J, Canal C, Cubo J. Issues in the formalization of Web Service Orchestrations. In: Proceedings of the Second International Workshop on Coordination and Adaptation Techniques for Software Entities(WCAT05), Spain, 2005

12 Foster H, Uchitel S, Magee J, et al. Model-based Verification of Web Service Compositions. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, USA, 2003

13 Victor B, Moller F. The Mobility Workbench-A Tool for the Pi-Calculus. In: Proceedings of the 6th International Conference on Computer Aided Verification, USA, 1994

14 Sangiorgi D. A Theory of Bisimulation for the pi-Calculus. In: Proceedings of the 4th International Conference on Concurrency Theory. Germany, 1993