第 31 卷 第 8 期
2008 年 8 月
计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS
Vol. 31 No. 8
Aug. 2008

基于适配的服务协作流程动态管理

乔晓强 魏 峻 黄 涛
（中国科学院软件研究所软件工程技术中心 北京 100190）

摘 要 服务协作以服务为构造单元，通过组合不同的服务，实现复杂的业务逻辑，并以此为基础构建新的应用. 为实现正确的服务协作，服务之间的交互必须是兼容且一致的. 适配技术通过构造适配器来弥补服务之间的交互差异，为解决协作的一致性问题提供了有效的方法途径. 然而考虑到服务计算环境的动态特性，协作适配器本身应具有动态调整及重新配置的能力，从而避免使其成为协作系统响应动态变化的可适应能力的瓶颈. 文中首先给出了一种基于适配的服务协作方法，以保证服务交互的一致性. 并针对提出的适配器结构，建立了适配器的执行模型，在此基础上给出了适配器的动态更新方法，使适配器具有可重配置以及动态调整的能力，从而保证了基于适配的服务协作系统的灵活性和动态可适应性.

关键词 Web 服务；服务协作；协议适配；动态管理
中图法分类号 TP311

A Dynamic Management Approach to Mediation-Based Service Collaboration Process

QIAO Xiao-Qiang WEI Jun HUANG Tao
(*Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing* 100190)

Abstract Service collaboration realizes complex business logic based on Web services and through which to construct applications. It is necessary of compatibility and consistency between interactive services for correct service collaboration. Mediation technologies provide effective approaches for bridging service differences. Considering the dynamic nature of service-oriented computing, mediation itself should be equipped with dynamic adaptation and re-configuration capabilities to avoid the bottle-neck of the dynamic management of collaboration system. This paper proposes a mediation-based approach to facilitate service collaboration when mismatches occur between interactive services. The execution model and the adaptation method for mediators are also presented to realize the re-configuration and dynamic adjustment of mediators at runtime, which ensures the agility and adaptability of service collaboration based on mediation mechanism.

Keywords Web services; service collaboration; protocol mediation; dynamic management

1 引 言

Web 服务作为自治的、自描述的和平台独立的模块化应用，具有开放的，面向 Internet 和基于标准的接口，能较好地解决异构应用之间、松散耦合环境下的互操作、集成和协作问题[1]. SOC(Service Oriented Computing)以服务为构造单元，通过 SOA 框

收稿日期：2008-04-13；最终修改稿收到日期：2008-06-20. 本课题得到国家自然科学基金(60673112)、国家“九七三”重点基础研究发展规划项目基金(2002CB312005)和国家“八六三”高技术研究发展计划项目基金(2006AA01Z163，2007AA010301)资助. **乔晓强**，男，1977 年生，博士研究生，主要研究方向为面向服务计算、工作流技术、服务协作理论. E-mail：qiaoxiaoqiang@otcaix.iscas.ac.cn. **魏 峻**，男，1970 年生，博士，研究员，博士生导师，主要研究领域为面向服务计算、中间件、移动计算和软件工程. **黄 涛**，男，1965 年生，博士，研究员，博士生导师，主要研究领域为软件工程和方法学、分布式对象技术.

架及相关的 SOAP、WSDL、UDDI 等标准研究服务的发布、协调以及交互，通过组合不同提供商的服务，来实现复杂的业务逻辑，并以此为基础构建新的应用，为企业间的 B2B 分布式协作提供了有效的解决方案[2-3].

在面向服务的企业应用集成中，服务通过相互的交互来实现业务目标流程，在实现服务协作时所需要考虑的问题包括：(1) 参与协作的服务之间交互一致性. 服务是由不同的提供商开发的，无法在设计阶段就预测出其潜在的所有交互可能，因此在实际的协作中，参与协作的服务会存在不一致性，因此无法保证正确、无缝的集成与协作. 这种不一致性不仅体现在静态方面(服务所支持的操作集合)，还体现在动态行为方面(服务所允许的消息交互的流程逻辑)[4-5]；(2) 服务组件的动态特性. 由于服务的自治性和松散耦合的特点，服务组件自身会升级演化，并改变其外部行为. 这些变化都会导致服务交互的不一致性，从而使原有的协作流程失效. 因此要求服务协作系统应具有动态可适应性，以应对各种变化.

服务适配技术为服务协作的一致性问题提供了解决方案. 文献[6]中提出服务的适配需求主要包含两类：兼容性与可替换性. 兼容性是指通过协调服务行为，使其可以与其它服务正确地交互. 可替换则是指通过适配器，协调一个服务的行为，使其与另一服务的行为相同，即可以实现对另一服务的替换，这样在服务由于升级或演化而改变其外部行为时，通过适配器可以屏蔽这种变化给协作流程所带来的影响. 而考虑到服务计算环境的动态变化性，服务的升级或者演化都会在协作实例运行时发生，因此适配器本身也应具有动态调整及重新配置的能力，从而避免使其成为协作系统响应动态变化的可适应能力的瓶颈.

针对上述问题，本文首先给出了一种基于适配器的服务协作方法，以保证服务与协作目标的一致性，并针对我们的适配器结构，给出了适配器的执行模型，在此基础上给出了适配器的动态更新方法，使适配器具有可重配置以及动态调整的能力，以保证基于适配的服务协作系统的灵活性和动态可适应性.

2　基于适配的服务协作

基于适配的服务协作是指通过适配器的建立，协调服务组件的行为，使得不兼容的服务之间可以正确地交互，其协作模型如图 1 所示.

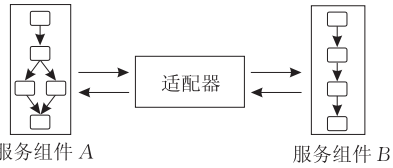


图 1　基于适配的服务协作模型

在给出适配器的构造方法之前，我们首先给出服务的行为接口定义，并结合实例进行说明. 这里我们使用 CSP 语言来严格地描述服务的行为. 本文中所用到的 CSP 的操作符子集主要包括：

$P = a \rightarrow P \mid \mu X \cdot F(X) \mid P \Box Q \mid P \sqcap Q \mid P;Q \mid P \parallel Q \mid$ SKIP | STOP；

$a \rightarrow P$：设有一事件为 a，一进程为 P，则执行事件 a，然后按照进程 P 的说明进行动作；

$\mu X \cdot F(X)$：$F(X)$ 是一包含了名称 X 的有前缀的流程描述，$\mu X \cdot F(X)$ 给出了循环执行流程 X 的定义；

$P \Box Q$：外部确定性选择(external deterministic choice)，即一个流程可以按照进程 P 的说明动作，也可按照 Q 的说明动作，其选择由外部环境决定；

$P \sqcap Q$：内部非确定性选择(internal non-deterministic choice)，即一个流程可以按照进程 P 的说明动作，也可以按照 Q 的说明动作，其选择由流程自身决定；

$P;Q$：表示流程首先按照进程 P 的说明动作，如果成功，则继续按照 Q 的说明动作；

$P \parallel Q$：表示通常的并行操作，进程 P 和 Q 将基于符号集中的公共事件进行同步，即进程 P 和 Q 在事件集 $\alpha P \cap \alpha Q$ 上同步；

STOP 表示流程死锁，SKIP 表示流程执行成功.

详细的操作语义可参见文献[7].

服务的接口描述了服务的功能以及如何使用该功能的方式，为服务协作和匹配提供基础，其行为兼容性除了要考虑单个服务方法调用的输入输出相匹配之外，还要全局考虑服务方法调用的时序关系是否满足正确性和一致性的要求. 因此我们基于服务交互过程中的消息流(对应于服务中方法的调用与返回)来描述服务组件的接口协议.

定义 1.　服务组件接口是一个三元组：$SI = (M_R, M_S, P)$，其中：

(1) M_R 是服务所有的接收消息集合，M_S 是服

务所有的发送消息集合；

(2) P 是服务组件的接口协议，P 的符号集是 $m!v:typeof(m)$ 和 $m?x:typeof(m)$ 的集合，其中 $m\in(M_R\cup M_S)$，用来标识通信的通道，$typeof(m)$ 表示消息 m 的类型.

$m!v:typeof(m)$ 表示在通道 m 上发送符合 m 的类型的消息 v，而 $m?x:typeof(m)$ 表示在通道 m 上接收符合 m 的类型的消息. 为了易于描述，我们将其分别表示为 $m!v$ 和 $m?x$. 服务之间的交互可以用操作符“‖”连接两个服务接口协议，则当一个服务协议活动为 $m!v$，而另一个服务协议的活动为 $m?x$ 时，双方执行一次通信操作，表示只有当交互一方处于消息发送状态，而另一方处于同一消息的接收状态时，实现一次正确的交互.

如图 2 所示，我们给出了一个在线购书的服务协作实例. 图中分别给出 Customer 与 Bookstore 服务的流程描述，其流程的 CSP 描述分别如下：

$P_C=\text{order}!v_o\rightarrow[(\text{acceptance}?x_a\rightarrow\text{shipment}?x_s\rightarrow\text{shipment_confirmation}!v_{sc}\rightarrow\text{payment}!v_p\rightarrow\text{payment_confirmation}?x_{pc}\rightarrow\text{invoice}?x_{in}\rightarrow\text{SKIP})\square(\text{rejection}?x_{re}\rightarrow\text{SKIP})]$；

$P_B=\text{order_info}?x_{oi}\rightarrow[(\text{OK}!v_{ok}\rightarrow[(\text{payment}?x_p\rightarrow\text{invoice}!v_{in}\rightarrow\text{SKIP})\parallel(\text{shipment_info}?x_{si}\rightarrow\text{shipment}!v_s\rightarrow\text{SKIP})]\rightarrow\text{SKIP})\sqcap(\text{Not_OK}!v_{nok}\rightarrow\text{SKIP})]$.

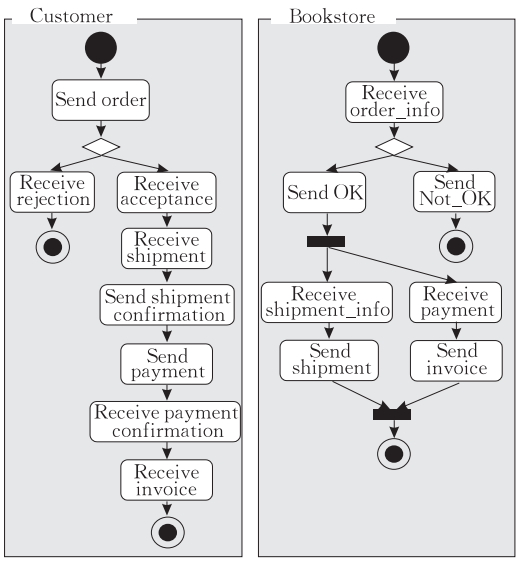


图 2 Customer 服务与 Bookstore 服务流程

两个服务除了在单个消息上存在不匹配(如 Customer 的 rejection、acceptance 消息与 Bookstore 的 OK 和 Not_OK 消息)之外，消息交互行为上也存在差异，被称为协议失配. 文献[6]提出，协议失配的类型包括顺序失配、冗余消息、缺失消息、消息合成与分解失配等等. 对于 Bookstore 服务而言，它与 Customer 服务之间的协议失配主要包括冗余的消息(shipment_confirmation)、缺失的消息(payment_confirmation)、消息分解(这里假设 Customer 服务的 order 消息包含了 Bookstore 服务的 order_info 和 shipment_info 两部分的消息)以及顺序失配(Customer 服务以顺序的流程来实现 payment 和 shipment 两类事件的交互，而 Bookstore 服务则并发地执行 payment 和 shipment 事件).

由于两个服务的交互协议存在不一致性，因此不能直接地交互，这些差异可以通过适配机制来弥补，从而保证协作的成功，提高服务的可重用性.

首先通过两个服务之间的数据依赖关系来建立消息的映射，即一方所需要的消息如何通过另一方所发送的消息来构造.

定义 2. 设需要适配的协议分别为 P_A 和 P_B，则它们之间的数据映射包含两类集合：$Map_{\langle A,B\rangle}$ 和 $Map_{\langle B,A\rangle}$. 其中 $Map_{\langle A,B\rangle}=\{synth_i(m_r,M_S)\mid m_r\in M_R^B, M_S\subseteq M_S^A, 1\leq i\leq n\}$，给出了从服务 A 的发送消息 M_S^A 到服务 B 的接收消息 M_R^B 的映射规则集合. M_S 是与 m_r 有依赖关系的发送消息集合. $synth$ 函数给出了基于 M_S 来构造消息 m_r 的方法.

同样，$Map_{\langle B,A\rangle}$ 是 M_S^B 到 M_R^A 的映射规则集合.

表 1 给出了 Customer 与 Bookstore 服务的数据依赖关系.

表 1 Customer 与 Bookstore 的数据依赖关系

Customer 服务消息		Bookstore 服务消息
order	←→	order_info
	←→	shipment_info
acceptance	←→	OK
rejection	←→	Not_OK
payment	←→	payment
shipment	←→	shipment
shipment_confirmation	←→	NULL
payment_confirmation	←→	NULL
invoice	←→	invoice

我们根据数据依赖关系构造数据映射规则. 例如，Bookstore 服务中 order_info 和 shipment_info 通过抽取 Customer 服务的 order 消息的相应部分得到，而对于缺少的 payment_confirmation 消息，则

通过构造一个常量的消息来实现.

适配器在建立数据映射规则的基础上进行构造. 当采用适配机制时，适配器在交互双方之间起到数据缓存、抽取、转换以及协调消息交互顺序的作用. 适配器可以接收任何一方所发送的消息，并且在需要的时候将消息发送到相应的接收方. 因此我们给出适配器的定义如下.

定义 3.　设需要适配的协议分别为 P_A 和 P_B，则 P_A 和 P_B 之间的适配器由一个三元组表示(P_A^M，P_B^M，$Pipes$)，其中：

(1) $Pipes$ 是消息管道集合，用来实现双方所交互数据的存储与转换. $Pipes=\{Pipe_m_i \mid 1\leq i\leq l\}$. l 是消息映射规则中所有接收消息的数目.

(2) P_A^M 和 P_B^M 分别实现与 P_A 和 P_B 的消息交互以及相应的数据存储与读取的流程.

适配器的构造方法如下：

(1) 对于数据映射规则中($synth(m_r, M_S)$)的每个接收消息 m_r，都存在一个对应的消息管道 $Pipe_m_r$，其行为描述如下：

$$Pipe_m_r=[(\parallel_{i\leq n}\text{left}?x_{si}: typeof(m_{si}))\rightarrow synth\rightarrow \text{right}!v: typeof(m_r)\rightarrow SKIP]\square[close\rightarrow SKIP].$$

这里 $m_{si}\in M_S$，n 是 m_r 所依赖的发送消息的数目. 消息管道用来缓存消息，它通过自己的 left 通道接收消息，并通过其 right 通道发送消息. $synth$ 是数据映射定义中的 $synth$ 函数的相应实现. 由于相对应的消息可能存在于选择分支中，因此在一次实际的交互过程中可能并不接收消息(选择了其它分支)，[$close\rightarrow$SKIP]的含义就是在交互过程全部完成时关闭消息管道. 如果消息存在于循环结构中，则其相应的行为描述为

$$Pipe_m_r=\mu X\cdot([(\parallel_{i\leq n}\text{left}?x_{si}: typeof(m_{si}))\rightarrow synth\rightarrow \text{right}!v: typeof(m_r)\rightarrow X]\square[close\rightarrow SKIP]).$$

(2) P_A^M 和 P_B^M 中包括消息接收和消息发送两类事件，分别通过将 P_A 和 P_B 中的事件进行如下替换来构造：

① $\forall m!v\in\alpha P_A$(或 αP_B)，如果 m 在数据映射中有相应的映射关系定义，则 P_A^M(或 P_B^M)中对应的消息接收事件为

A(或 B)$.m?x\rightarrow(\parallel_{i\leq n}Pipe_m_i.left!x)$，$Pipe_m_i$ 是依赖于 m 的接收消息对应的消息管道；通道前加上服务名称以区分各自的交互对象；

否则，P_A^M(或 P_B^M)中对应的事件为 A(或 B). $m?x\rightarrow$SKIP.

其功能是接收消息 m，并写入与 m 有数据依赖关系的消息管道.

② $\forall m?x\in\alpha P_A$(或 αP_B)，若 m 在数据映射中有相应的映射关系定义，P_A^M(或 P_B^M)中对应的消息发送事件为 $Pipe_m.right?x\rightarrow A$(或 B)$.m!x$；

否则，P_A^M(或 P_B^M)中对应的事件为 STOP.

其功能是读取消息管道中存储的消息 m，并发送给相应的服务.

(3) 适配器的行为($\parallel_{i\leq n}Pipe_m_i$)$/\!/[P_A^M\parallel P_B^M]$.

$P_A^M\parallel P_B^M$ 并发执行，通过消息管道实现两者之间的数据交换. 符号"//"是从属操作符. "//"符号左边的部分如同 $P_A^M\parallel P_B^M$ 的附属流程. 附属流程中的事件只有在主流程 $P_A^M\parallel P_B^M$ 允许其发生的时候，才可以发生.

适配器的正确性是对($\parallel_{i\leq n}Pipe_m_i$)$/\!/[P_A^M\parallel P_B^M]$结构进行检查. 如果流程($\parallel_{i\leq n}Pipe_m_i$)$/\!/[P_A^M\parallel P_B^M]$不存在死锁，构造的适配器是有效的，否则 P_A 和 P_B 之间不存在有效的适配器. 详细的检查算法请参见文献[8]. 适配器的 P_A^M 和 P_B^M 部分自动构造算法如下.

算法 1.　适配协议的自动构造.

输入：协议 p，服务名称 r

输出：p 所对应的适配协议部分 p'

```
p' := copy(p)
for every activity a∈αP'
  if (a.m∈r.M_R)   /**a 为接收消息事件**/
    if (∃synth_i(m_r,M_S)∈Map∧a.m=m_r)
    b := union ("→",
      new ReceiveActivity(Pipe_m.right,m),
      new SendingActivity(r,m));
    else b := STOP;
    replace(p',a,b);
    end if
  else if (a.m∈r.M_S)   /**a 为发送消息事件**/
    b :={};
    for all synth_i(m_r,M_S)∈Map
      if (a.m∈M_S)
        b := union ("∥", b,
          new SendingActivity(Pipe_m_r.left,m));
      end if
    end for
    if (b!={})
    b:=union("→",new ReceiveActivity(r,m),b);
  else b:=union("→",new ReceiveActivity (r,
          m),SKIP);
```

```
20.     replace (p',a,b);
21.     end if
22.   end if
23. end for
```

算法的输入为需要适配的服务接口协议 P 及服务名称 r. *ReceiveActivity* 和 *SendActivity* 方法用来构造新的接收和发送活动. 构造 P_A^M 和 P_B^M 的总时间复杂度为 $O(m+n)$, m 和 n 分别为 P_A 和 P_B 的活动数目.

在协议适配的相关工作方面,文献[6,9]只是分析了服务接口之间的协议失配形式,没有给出适配器的自动构造方法. 文献[10]使用有限状态机对组件接口协议进行建模,其局限性在于描述方法表达能力不足,例如无法表述确定性选择、非确定性选择、流程的并发组合等等,因此需要人工加入额外的映射关系描述,来完成适配器的构建. 而且该方法基于状态空间进行多步的迭代化简求解,当服务的交互变得复杂的时候,会出现状态空间过大,从而导致方法的复杂度增大. 文献[11]使用 Pi-Calculus 对接口协议建模,具有相当的表达力,然而该方法通过逐步添加活动的方式来实现适配器的构造,每一步都需要对已有交互结构进行一次死锁检查,而且没有给出处理并发和选择结构的方法. 我们的方法采用 CSP 这种表达力强,而且有效刻画交互行为的形式化方法,提高协议和适配器的描述能力,从而无需附加额外的映射关系描述;同时基于原有的协议结构来构建适配器的协议结构,只需要对整体交互结构作一次死锁检查,方法更有效且易于实现.

对于服务 Customer 与 Bookstore,我们所得到的适配器的 P_C^M 和 P_B^M 如下(服务名称分别设为 C 和 B):

$$
\begin{aligned}
P_C^M = {} & C.order?x_o \rightarrow [Pipe_oi.left!x_o \parallel \\
& Pipe_si.left!x_o] \rightarrow [(Pipe_ac.right?x_{ac} \rightarrow \\
& C.acceptance!x_{ac} \rightarrow Pipe_s.right?x_s \rightarrow \\
& C.shipment!x_s \rightarrow C.shipment_confirmation?x_{sc} \rightarrow \\
& C.payment?x_p \rightarrow Pipe_p.left!x_p \rightarrow \\
& Pipe_pc.right?x_{pc} \rightarrow C.payment_confirmation!x_{pc} \rightarrow \\
& Pipe_in.right?x_{in} \rightarrow C.invoice!x_{in} \rightarrow \mathrm{SKIP}) \square \\
& (Pipe_re.right?x_{re} \rightarrow C.rejection!x_{re} \rightarrow \mathrm{SKIP})],
\end{aligned}
$$

$$
\begin{aligned}
P_B^M = {} & Pipe_oi.right?x_{oi} \rightarrow B.order_info!x_{oi} \rightarrow \\
& [(B.OK?x_{ok} \rightarrow Pipe_ac.left!x_{ok} \rightarrow \\
& [(Pipe_si.right?x_{si} \rightarrow B.shipment_info!x_{si} \rightarrow \\
& B.shipment?x_s \rightarrow Pipe_s.left!x_s \rightarrow \mathrm{SKIP}) \parallel \\
& (Pipe_p.right?x_p \rightarrow B.payment!x_p \rightarrow \\
& B.invoice?x_{in} \rightarrow Pipe_in.left!x_{in} \rightarrow \mathrm{SKIP})]) \square \\
& (B.Not_OK?x_{nok} \rightarrow Pipe_re.left!x_{nok} \rightarrow \mathrm{SKIP})].
\end{aligned}
$$

3 适配器的执行模型

适配器的流程定义是对适配器的交互行为的描述,适配器的执行模型则描述了一个适配器的执行方式. 我们的方法所给出的适配器定义为 $(\parallel_{i \le n} Pipe_m_i) // [P_A^M \parallel P_B^M]$,其特点在于 P_A^M 和 P_B^M 作为主流程实现与实际服务的消息交互,而 $\parallel_{i \le n} Pipe_m_i$ 作为消息的存储、抽取与转换进程,并发地来执行操作. 因此在适配器的执行模型中,活动中所处理的消息均用变量(*variable*)来表示,P_A^M 和 P_B^M 作为并发的流程,实现消息的收发和变量的存取操作;每个 *Pipe_m* 实现为独立的活动,对输入变量进行抽取与转换操作,并生成输出变量. 其与主流程之间的交互依赖关系通过同步控制(synchronization)机制来实现.

我们首先给出执行模型中的活动(*activity*)的基本结构描述,然后给出适配器执行实例的定义. 一个适配器执行实例的活动分为基本活动和数据转换活动两类,其结构信息分别如表 2 和表 3 所示.

表 2 基本活动的结构信息

名称	描述
activityID	活动的标识
type	活动类型 {*RECEIVE*,*SEND*}
variable	所对应的消息变量
preActivities	当前活动的前置基本活动集合
succActivities	当前活动的后继基本活动集合
synchronizers	活动所对应的需同步的数据转换活动的集合
Status	活动状态{*NOT_ACTIVATE*,*ACTIVATED*,*RUNNING*,*COMPLETED*,*SUSPENDED*}

表 3 数据转换活动的结构信息

名称	描述
activityID	活动的标识
inputVariables	所对应的输入消息变量集合
outputVariable	所对应的输出消息变量
inputSynchronizers	所对应的接收活动的集合
outputSynchronizer	所对应的发送活动
status	活动状态{*ACTIVATED*,*COMPLETED*,*CANCELLED*}

基本活动包含了发送和接收两类活动,分别对应于定义中的消息接收事件和消息发送事件.

synchronizers 是与基本活动有依赖关系的数据转换活动,对于消息发送活动,其 *synchronizers* 是输出变量为其对应消息变量的数据转换活动. 而对于消息接收活动,其 *synchronizers* 是所有输入变量为其对应的消息变量的数据转换活动集合.

从表 2 中可以看出,基本的活动状态包括有

{*NOT_ACTIVATE*, *ACTIVATED*, *RUNNING*, *COMPLETED*, *SUSPENDED*}. 其状态转换如图 3 所示. 对于一个基本活动，在流程启动时进入 *NOT_ACTIVATE* 状态，在满足启动条件后，进入 *ACTIVATED* 状态. 对于顺序结构中的活动，其启动条件为前置活动进入 *COMPLETED* 状态；并发结构后的活动启动条件：所有的前置活动全部处于 *COMPLETED* 状态；分支结构后的活动启动条件：任意一个前置活动进入 *COMPLETED* 状态.

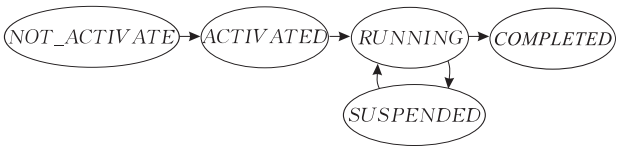


图 3　基本活动的状态转移图

处在 *ACTIVATED* 状态的活动，在其所依赖的同步数据转换活动完成之后，进入 *RUNNING* 状态. 完成之后进入 *COMPLETED* 状态. *SUSPENDED* 状态则用来暂停处于 *RUNNING* 状态的活动.

从表 3 中可以看出，数据转换活动的状态则有 {*ACTIVATED*, *COMPLETED*, *CANCELLED*}. 数据转换活动在流程启动时进入 *ACTIVATED* 状态，在所依赖的基本活动均进入状态 *COMPLETED* 之后，执行数据转换，并进入 *COMPLETED* 状态. 流程结束的时候，所有处于 *ACTIVATED* 的数据转换活动均被取消，状态设置为 *CANCELLED*. 其转换图如图 4 所示.

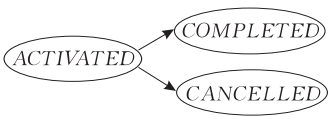


图 4　数据转换活动的状态转移图

变量 *variable* 则表示为(id, $type$, $value$)，id 为 *variable* 的标识，$type$ 是其类型，$value$ 则是 *variable* 的值.

适配器的执行实例则定义如下.

定义 4.　一个适配器实例表示为一个三元组：$I_M=(id, M, S_M)$，描述了一个运行中的适配器的即时状态，其中：

(1) id 是适配器实例的标识. 一个适配器定义对应了多个实例，id 用来唯一标识一个具体的实例；

(2) M 是实例所对应的适配器的定义；

(3) S_M 是一个适配器实例的运行状态，由适配器定义中所有活动状态的集合来描述，由一个四元组表示：(T_A, T_B, U, V)，其中：

(i) T_A 是适配器定义中 P_A^M 中所有基本活动的集合；同样的 T_B 是 P_B^M 中所有的基本活动的集合；

(ii) U 是所有的数据转换活动的集合；

(iii) V 是适配器中所有变量的集合.

4　适配器的实例迁移

在协作服务的协议行为发生变化的时候，需要重新生成适配器，因此适配器的行为结构会发生变化，此时需要将处于运行状态的适配器实例暂停，并按照新的适配器定义进行执行，称作适配器的实例迁移，其定义如下.

定义 5.　适配器定义的动态变化为 $\delta_M=(M, M')$，即定义由 M 转化为 M'. 对于一个适配器实例 $I_M=(id, M, S_M)$，其迁移表示为 $\delta_S=(S_M, S'_M, \delta_M)$，则变化后的实例为 $I'_M=(id, M', S'_M)$.

在实现适配器的实例迁移时，首先对变量的值和活动的状态进行迁移.

4.1　变量的迁移规则

变量的迁移规则如下所示.

规则 1.　$\forall variable \in I'_M.S'_M.V$，如果：

$$variable \in I_M.S_M.V \wedge ValueOf_{IM}(variable)=v,$$

则 $variable.value := v$，否则 $variable :=$ NULL.

规则 1 的作用在于，当新流程中的变量存在于原流程实例定义中时，复制其值. 其中 $ValueOf_{IM}$ 的作用在于取变化前实例中变量的值.

4.2　活动的状态迁移规则

对于 $\forall activity \in I'_M.S'_M.T$（这里 $T=I_M.S_M.T_A \cup I_M.S_M.T_B$），其状态迁移规则如表 4 所示.

表 4　规则 2 基本活动的状态迁移规则

条件	*activity* 的状态设置
$activity \notin I_M.S_M.T$	*NOT_ACTIVATE*
$activity \in I_M.S_M.T \wedge StateOf_{IM}(activity)=$ *NOT_ACTIVATE*	*NOT_ACTIVATE*
$activity \in I_M.S_M.T \wedge StateOf_{IM}(activity)=$ *ACTIVATED*	*NOT_ACTIVATE*
$activity \in I_M.S_M.T \wedge StateOf_{IM}(activity)=$ *RUNNING*	*SUSPENDED*
$activity \in I_M.S_M.T \wedge StateOf_{IM}(activity)=$ *COMPLETED*	*COMPLETED*

对于数据转换活动，其状态主要根据基本活动的状态来确定.

规则 2.　$\forall activity \in I'_M.S'_M.U$.

如果该活动所对应的 *inputSynchronizers* 活动状态均为 *COMPLETED*，则

$$activity.status := COMPLETED;$$

否则，$activity.status := ACTIVATED$.

4.3 迁移后的适配器实例的执行

上述规则只是依据历史执行信息静态地确定了活动状态，即是否已完成，是否未执行等等. 要实现迁移后的适配器实例的执行，还需要在此基础上进一步确定转换后得到的适配器实例的执行方式，即从哪些活动开始恢复适配器的执行.

我们依据当前活动所标记的状态，计算出迁移后适配器实例的可执行部分(execution part). 具体如算法 2 所示. 在执行算法前，首先对 $P_A^{M'}$ 和 $P_B^{M'}$ 进行预处理，将并发和选择结构前后的"→"操作符等价转换为操作符";"，这样流程转换为如下形式：$P_1;P_2;\cdots;P_n$. 其中 $P_1,P_2,\cdots,P_n$ 等均为并发、选择或形如 $activity_1 \rightarrow activity_2 \rightarrow \cdots \rightarrow activity_n$ 的结构.

算法 2. 迁移后适配器实例的可执行流程.

```
Procedure executionPart
输入：适配器实例协议 p，所得到的可执行流程 q
输出：是否完成标志 isExit
 1. if isSequential(p)
                    /**存在多个顺序流程，反向遍历**/
 2.   for (i := n; i>0;i--) do
 3.     p_i := subSequentialProtocol(p,i);
 4.     isExit := executionPart(p_i,q_i);
 5.     q := union(q_i,q,";");
 6.     if (isExit) return isExit; end if
 7.   end for
 8.   return false;
 9. else if(isParallel(p))   /**并发结构**/
10.   for all subParallelProtocol p_i of p do
11.     if (executionPart(p_i,q_i)) isExit := true; end if
12.     q := union(q,q_i,"||");
13.   end for
14.   return isExit
15. else if isChoice(p)   /**选择结构**/
16.   for all subChoiceProtocol p_i of p do
17.     if (executionPart(p_i, q_i)) q := q_i; return true;
18.     end if
19.   end for
20.   q=p;
21.   return false
22. else   /**形如 a_1→a_2→…→a_n的结构**/
23.   for (i := n; i >0; i--) do
24.     if (a_i.status=COMPLETED)return true;
25.     end if
26.     q := union(q,a_i,"→");
27.   end for
28.   return false;
29. end if
```

算法 2 所得到的部分也是一个可执行流程，适配器将基于此流程恢复其执行. 对于可执行流程中的起始活动，若其状态为 *NOT_ACTIVATE*，则将其状态设置为 *ACTIVATED*，而状态为 *SUSPENDED* 的活动，状态则转换为 *RUNNING*. 对应于 $P_A^{M'}$ 和 $P_B^{M'}$，其可执行部分的流程分别定义为 $P_{AE}^{M'}$ 和 $P_{BE}^{M'}$. 此算法的基本思想是对变化后的适配器定义中的 $P_A^{M'}$ 和 $P_B^{M'}$，分别从其终结活动开始，反向遍历其活动节点，若活动的状态为 *NOT_ACTIVATE* 或者 *SUSPENDED*，满足启动条件而且其后续的活动节点中不存在状态为 *COMPLETED* 的活动，则当前活动作为可执行流程的起始节点，并停止遍历过程，返回当前得到的流程作为可执行流程. 算法的 1～8 行处理顺序子流程，若存在满足条件的子流程，则顺序连接其后的子流程并返回；9～14 行处理并发子流程，若存在一个满足条件的子流程，则合并所有子流程的计算结果，并返回；15～21 行处理选择结构，若存在一个满足条件的子流程，则返回该子流程的计算结果，否则返回整个选择结构；算法的最后部分处理基本活动的顺序流程，同样采用反向遍历，当查找的活动状态为 *COMPLETED* 时，返回当前得到的流程. 算法的时间复杂度为 $O(n)$，n 为协议的活动数目.

4.4 迁移实例执行的正确性检查

在完成适配器实例状态的动态迁移之后，还需要对其正确性进行检查，即 4.3 节所得到的迁移后适配器实例的可执行流程是否可以正确地执行.

适配器的正确性检查包括两部分：适配器定义的正确性检查以及实例执行状态的正确性检查. 适配器定义的正确性检查主要考虑交互的服务双方之间是否存在有效的适配器(结构中不存在死锁)，这一步检查工作可以在重新构造适配器时进行.

适配器定义的正确性并不能保证当前实例在恢复执行后能够正确地实现与协作服务的消息交互，因此还需要针对当前实例的执行状态进行正确性检查. 即在当前的状态下，若恢复适配器实例的执行，那么对于服务双方所需要的消息，能否由实例当前的可执行部分构造而得到. 实例执行状态的正确性检查主要包括以下两个步骤：

(1) 首先要在迁移后的实例中，查找所有状态为 *NOT_ACTIVATE* 或 *SUSPENDED*，而不在可

执行部分的活动，将其状态设置为 $SKIPPED$. 即

$$\forall activity \in I'_M.S'_M.T \wedge StateOf_{IM'}(activity) = (NOT_ACTIVATE \mid SUSPENDED) \wedge activity \notin P^{M'}_{AE} \cup P^{M'}_{BE},$$

则，$activity.status := SKIPPED$.

上述所得到的活动由于其后继活动中存在状态为 $COMPLETED$ 的活动，因此它们不属于可执行部分，从而无法在当前适配器实例中执行. 这里我们引入临时状态 $SKIPPED$ 来表明活动被跳过而未执行.

如果处于 $SKIPPED$ 的活动为消息发送活动，且不位于未被选择的选择分支中，则说明实际服务所需的消息不会被构造，因而会引起交互错误，即适配器实例无法正确地迁移.

(2) 对于每一个可执行部分的发送消息活动，必须能够正确地构造所需的消息并发送给相应的服务. 具体的检查方法如下：

对于 $activity.type = SEND$ 的活动，设 $activity'$ 为 $activity$ 所依赖的数据转换活动，即 $activity' = activity.synchronizers$.

$$activity'.status = ACTIVATED \tag{1}$$

$$\exists a \in activity'.inputSynchronizers$$

$$a.status = SKIPPED \tag{2}$$

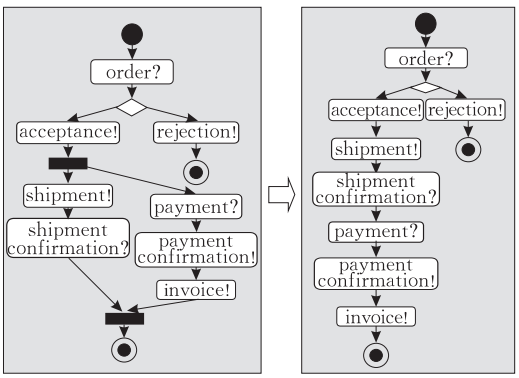
$$activity.variable = \mathrm{NULL} \tag{3}$$

当 $\exists activity \in P^{M'} \cup P^{M'}_{BE}$，同时满足上述 3 个条件时，说明该活动无法在实际运行中构造所需的消息. 对于消息发送活动，条件(1)说明了该活动所依赖的数据转换活动未完成，条件(2)则指明了该消息所依赖的接收消息活动中，存在未执行但是将会跳过的活动，因此数据转换活动无法在以后的执行中进入 $COMPLETED$ 状态. 而条件(3)则说明了该活动所需发送的消息在先前实例运行过程中也并未构造. 这样，该活动在实际运行中就无法构造和发送实际服务所需要的消息，因而导致协作的失效. 若 $P^{M'}_{AE} \cup P^{M'}_{BE}$ 中不存在满足上述条件的活动，则说明转换后的适配器实例可以正确地实现与服务的交互. 这里我们省略具体的实现算法描述. 接下来将结合具体的例子进行说明.

5　实例验证与分析

结合我们前面给出的在线购书的协作例子，我们对上一节给出的适配器实例迁移的正确性检查方法进行了验证，并分析说明算法是如何应用在具体的交互实例之中的.

假设 Customer 的适配协议的变化如图 5 所示，如果当前适配器实例状态为 shipment!v_s 活动处于激活($ACTIVATED$)状态，而 payment?x_p 活动处于完成($COMPLETED$)的状态，则算法返回的结果为 false. 这是因为针对上述实例进行状态迁移后，所得到的 Customer 服务所对应的适配器协议的可执行流程为 payment_confirmation!v_{pc} → Invoice!v_{in}，而 shipment!v_s 活动和 shipment_confirmation?x_{sc} 活动的状态则为 $SKIPPED$ 状态. 这时进行正确性检查，由于 shipment!v_s 为发送活动，而其状态为 $SKIPPED$，则说明无法正确地迁移适配器实例. 对应实际变化后的 Customer 服务来看，在收到送货消息之前，不应该执行付款服务，因此状态交互存在不一致性.



(a) 变化前　　(b) 变化后

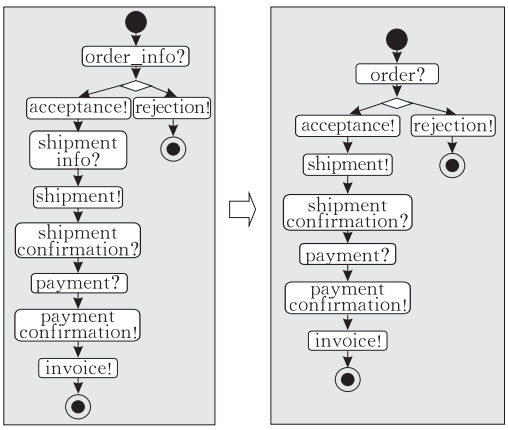
图 5　Customer 适配器协议动态迁移(1)

而如果当前适配器的状态为 shipment!v_s 活动和 payment?x_p 活动均处于完成($COMPLETED$)的状态，则算法返回结果为 true. 首先状态迁移后所得到的可执行流程与前一个例子相同，而只有 shipment_confirmation?x_{sc} 活动为 $SKIPPED$，此时由于 Bookstore 服务(见图 2)对应的适配器协议部分的消息都不依赖于 shipment_confirmation 消息，则无论 Bookstore 服务对应的适配器协议处在任何状态，都可以顺利地执行，因此适配器实例可以正确的迁移.

对于处于其它状态下的适配器状态，我们可以看到，这种结构上的变化均不会造成协作的失效，因此适配器实例总可以正确地迁移.

又如图 6 所示，Customer 服务变化前后对应的适配器协议分别如(a)和(b). 此时若适配器实例状态为 shipment_info?x_{si} 活动处于 $COMPLETED$ 状态. 此时算法的返回结果为 true. 通过分析可以看出，迁移后 shipment_info?x_{si} 活动被删除，那么对于

在 Bookstore 服务的适配器流程中的 shipment_info!v_{si}活动，其在新的适配器定义中所依赖的 order?x_o活动是新增活动，而且其状态为 *SKIPPED*，但是由于 shipment_info!v_{si}所依赖的 shipment_info 变量并不为 NULL，说明该活动所需的数据已经在历史交互过程中得到，因此正确性检查的结果表明实例可以正确地实现动态迁移. 而如果变化发生时 shipment_info?x_{si}活动处于 *ACTIVATED* 状态，则算法返回 false，因为在种情况下活动 shipment_info!v_{si}就无法合成 shipment_info 消息，从而导致协作的失效.



(a) 变化前 (b) 变化后

图 6 Customer 适配器协议动态迁移(2)

另外，在前述的两个例子中，如果当前实例运行到 rejection!v_{re}分支，那么算法返回结果总为 true. 这是因为上述这些流程结构的变化均不会影响到 rejection!v_{re}分支，因此总能够正确地实现状态的动态迁移.

从这两个例子我们可以看出，由于适配器主要功能是实现与协作双方的数据交互与协调，因此这里我们主要从数据的角度来考虑交互正确性. 适配器实例迁移的正确性检查方法首先检查是否存在无法执行的消息发送活动(如例 1 中的 shipment!v_s活动)，以确定是否服务组件所需的消息都可以正确地发送. 接下来检查每条可执行路径是否都可以正确地实现与协作服务的交互，即服务组件所需要的消息是否都可以正确地合成，其所依赖的数据可以是先前接收到的消息(如例 2 中的 shipment_info 消息)，也可以是在接下来的交互过程所接收到的数据. 同时适配器的目的是保证协作双方的交互的正确性，因此对已执行部分的修改(例 1 中的 shipment_confirmation?x_{sc}活动以及例 2 中的 order_info?x_{oi})，只要不影响所需消息的合成，都可以保证协作实例的正确迁移.

6 原型实现

根据本文所提出的基于适配的服务协作流程动态管理方法，我们进行了原型系统的开发与实现，通过提供相应的开发环境以及运行时管理模块，实现适配器的自动生成、部署和运行，同时提供了支持运行时适配器动态调整的管理功能. 原型系统的体系结构如图 7 所示，其中开发环境部分基于 Eclipse 框架实现. 运行管理部分则主要基于中国科学院软件研究所开发的业务流程管理系统 OnceBPEL 来实现，OnceBPEL 系统的功能主要是实现基于 WS-BPEL 语言的业务流程的解析与执行以及业务流程实例的运行时管理. 我们扩展了 OnceBPEL 的相关功能模块来支持适配器的动态管理.

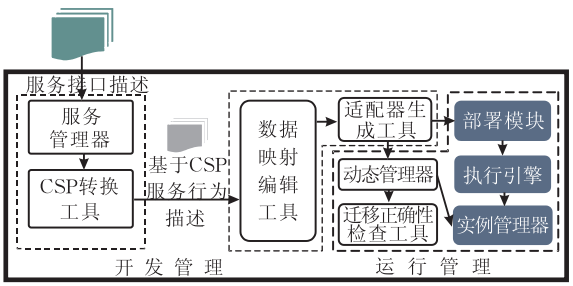


图 7 实现原型的体系结构

服务管理器负责管理参与协作的服务，服务的行为接口通过 BPEL 抽象流程来描述. BPEL 抽象流程可以有效地描述服务交互协议而且不展现内部细节. CSP 转换工具将 BPEL 抽象流程转换成为 CSP 的描述形式. 在转换时，我们依据协作角色信息(partnerlink)，只抽取需要与当前对应的服务进行交互的流程部分. 其转换规则如表 5 所示.

数据映射编辑工具用来可视化地编辑消息之间的映射规则. *synth* 函数通过 XSLT 语言来实现数据的抽取与转换. 适配器流程由适配器生成工具自动地构造. 部署模块则负责将适配器部署到执行引擎.

动态管理器实现适配器流程的动态管理，迁移正确性检查工具则根据输入的实例状态进行正确性检查. 引擎的实例管理器实现运行时的实例管理，包括流程的启动、挂起、恢复和停止以及获取实例状态等功能.

在实际应用的开发阶段，首先 CSP 转换工具将服务管理器所获得的服务组件的接口协议转换为 CSP 的描述形式，并输入到数据映射编辑工具，通过数据映射编辑工具实现服务组件之间数据映射规

表 5　BPEL 活动与 CSP 的映射规则

WS-BPEL	CSP
⟨receive varabile="*va*" … /⟩	$va?x$
⟨reply varabile="*va*" … /⟩	$va!v$
⟨invoke inputvariable="*vi*" outputvariable="*vo*" … /⟩	$vi!v \rightarrow vo?x$
⟨empty … /⟩	$SKIP$
⟨terminate … /⟩	$STOP$
⟨sequence …⟩ *activity* P_1 … *activity* P_n ⟨sequence/⟩	$P_1;\cdots;P_n$
⟨flow …⟩ *activity* P_1 … *activity* P_n ⟨flow/⟩	$P_1 \parallel \cdots \parallel P_n$
⟨pick…⟩ ⟨onMessage variable="v_1" …⟩ *activity* P_1⟨onMessage/⟩ … ⟨onMessage variable="v_n" …⟩ *activity* P_n⟨onMessage/⟩ ⟨pick/⟩	$(v_1?x_1 \rightarrow P_1) \square \cdots \square (v_n?x_n \rightarrow P_n)$
⟨switch…⟩ ⟨case…⟩ *activity* P_1⟨case…/⟩ … ⟨case…⟩ *activity* P_n⟨case…/⟩ ⟨switch/⟩	$P_1 \sqcap \cdots \sqcap P_n$
⟨while …⟩ *activity* P ⟨while/⟩	$\mu X \cdot (P;X \sqcap SKIP)$

则的可视化编辑. 接下来适配器生成工具模块将检查是否存在有效的适配器，并依据服务接口信息将适配器转换为相应的 BPEL 流程. 最后通过部署模块部署到执行引擎.

适配器实现为可执行的 BPEL 流程，P_A^M 和 P_B^M 分别实现为 flow 结构的两个并发子流程，数据转换活动则实现为 flow 的子活动，它们与基本活动之间的同步约束则通过 link 元素来实现.

在运行阶段，服务组件通过适配器的协调作用实现相互之间的协作. 如图 8 所示，我们给出了在运行期间，当服务组件的接口发生动态变化时，各个模块之间协作的时序图. 首先服务管理器接收到新的接口协议的定义，然后适配器生成工具转换生成新的适配器流程. 动态管理器则通过实例管理器暂停当前适配器流程的执行，并将新的适配器定义与当前的流程状态发送给迁移正确性校验模块，进行实例迁移的正确性校验. 在可以进行动态迁移的情况下，通过实例管理器更新当前的适配器流程实例，并恢复其执行；否则将失效信息返回给用户. 用户可以根据实际的应用情况，或者选择取消当前适配流程的执行；或者继续按照原有的适配器定义执行(需要服务提供者支持多版本服务组件的运行).

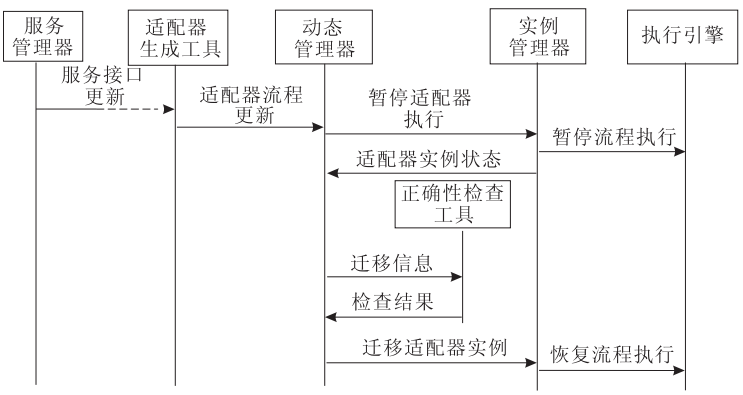


图 8　运行时系统的协作时序图

7　相关工作

在针对组件以及服务的适配研究方法中，文献[6,9]分析服务接口之间的协议失配形式. 文献[10-11]分别使用有限状态机以及 Pi-Calculus 对组件接口协议进行建模，实现适配器的自动构造. 与这两种工作相比，我们的工作在适配器自动构造的基础上，给出了运行时适配器的动态重配方法. 文献[12]提出了基于适配来增量式的构造协作系统的方法，并描述了适配器的动态配置问题. 文献[13]提出了依赖于反射机制的自适应服务的概念来实现适配服务的动态更新，然而上述两种方法只是给出了解决问题的基本途径，并没有给出适配器的构造算法、结构描述以及动态重配的具体实现方法.

在流程动态更新及实例迁移方面，传统的工作流技术领域有着众多的研究工作. 具体可以分为：

(1) 基于图形等价性的方法，例如文献[14]基于 WF Nets 来描述流程，并给出了实例动态调整的实现方法，要求变化前后的流程结构应具有继承关

系，并给出了状态转换规则，但其正确性检查并不考虑实例的实际执行情况. $WASA_2$[15]和 MILANO[16]则要求已执行的流程结构在变化后的流程定义中具有相对应的部分(包括活动和链接)，其允许迁移的条件过于严格.

(2)基于执行轨迹等价性的方法，例如 FLOW Nets[17]，TRAM[18]，WIDE[19]以及 $ADEPT_{flex}$[20]等等，动态迁移的正确性主要依据流程已执行轨迹是否可以在变迁后流程定义中复制来进行检查. 其中[19-20]还提供了基于变化元操作(例如添加/删除一个活动，添加/删除一个变迁，并发活动串行化等等)进行正确性检查的条件.

我们提出的适配器流程与传统的工作流具有不同的特点，因此与上述工作相比，我们的方法特点在于：首先，我们的方法在整体上对适配器实例的状态进行直接的迁移与检查，而不是依据于变化的元操作[19-20]来实现；一方面由于适配器是依据协作服务的行为自动生成，在发生动态调整时，难于确定具体是应用了哪些变化元操作；另一方面引入元操作变化容易造成状态迁移的不连续性，从而影响迁移条件的正确性检查；

其次，适配器主要实现服务之间的数据交互，因此我们提出的实例迁移方法不仅考虑控制流的变化，而且考虑了数据的依赖关系的变化，不同于文献[14-18]等只是从控制逻辑的改变方面来解决实例的动态调整问题；

再次，文献[15-16，18-20]中的方法都不允许改变流程的已执行部分. 而对于适配器流程而言，其主要作用是实现协作双方的交互与协调，重点考虑与服务的交互关系，因此只要保证交互的正确性. 从我们第 5 节的实例分析中可以看出，我们的方法允许对流程的已执行部分进行改变，只要这种改变不会造成动态改变后适配器与协作服务交互的失效.

8 结束语

本文提出了一种基于适配器的服务协作方法，以保证服务与协作目标的一致性. 针对我们的适配器结构，实现了适配器的动态配置，并给出了适配器动态调整的正确性检查方法，从而使适配器具有可重配置以及动态调整的能力，保证了基于适配的服务协作系统的灵活性和动态可适应性，方法中所提出的状态迁移规则和条件也易于验证和实现.

我们将在以下几个方面展开下一步的研究工作. 首先从事务方面进一步考虑适配器的构造方法，以保证服务的事务特性，并利用服务操作的事务特性，在适配器无法正确进行动态迁移时，通过回滚操作等方法使系统进行回滚，从而进入一个可以正确实现迁移的状态. 其次在建立数据映射关系方面，如何利用自动模式匹配技术以提高其自动化程度是我们需要考虑的另一个方向.

参 考 文 献

[1] Papazoglou M P, Georgakopoulos D. Service oriented computing. Communication of the ACM, 2003, 46(10): 24-28

[2] Benatallah B, Medjahed B, Bouguettaya A, Elmagarmid A, Beard J. Composing and maintaining Web-based virtual enterprises//Proceedings of the 1st Workshop on Technologies for E-Services (TES'00). Cairo, Egypt, 2000: 155-174

[3] Sheng Q Z, Benatallah B, Dumas M, Mak E O-Y. SELF-SERV: A platform for rapid composition of Web services in a peer-to-peer environment//Proceedings of the 28th VLDB Conference. Hong Kong, China, 2002: 1051-1054

[4] Christophides V, Hull R, Karvounarakis G, Kumar A, Tong G, Xiong M. Beyond discrete e-services: Composing session-oriented services in telecommunications//Proceedings of the 2nd Workshop on Technologies for E-Services (TES'01). Rome, Italy, 2001: 58-73

[5] Benatallah B, Casati F, Toumani F. Web services conversation modeling: A cornerstone for E-Business automation. IEEE Internet Computing, 2004, 8(1): 46-54

[6] Benatallah B, Casati F, Grigori D, Nezhad H R M, Toumani F. Developing adapters for Web services integration//Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05). Porto, Portugal, 2005: 415-429

[7] Hoare C A R. Communicating Sequential Processes. New Jersey, USA: Prentice Hall, 1985

[8] Qiao X Q, Wei J. Technical report: A decentralized services choreography approach based on mediation. Technology Report: TR-07-24, Institute of Software, Chinese Academy of Sciences, 2008

[9] Cimpian E, Mocan A. WSMX process mediation based on choreographies//Proceedings of the 1st International Workshop on Web Service Choreography and Orchestration for Business Process Management. Nancy, France, 2005: 130-143

[10] Yellin D M, Storm R E. Protocol specification and component adaptors. ACM Transactions on Programming Languages and Systems, 1997, 19(2): 292-333

[11] Bracciali A, Brogi A, Canal C. A formal approach to component adaptation. Journal of Systems and Software, 2005, 74(1): 45-54

[12] Poizat P, Salaün G, Tivoli M. On dynamic re- configuration of behavioral adaptations//Proceedings of the 3rd International Workshop on Coordination and Adaptation for Software Entities(WCAT′06). Nantes, France, 2006: 61-69

[13] Heuvel W, Weigand H, Hiel M. Configurable adapters: The substrate of self-adaptive web services//Proceedings of the 9th International Conference on Electronic Commerce (ICEC′07). Minneapolis, USA, 2007: 127-134

[14] Aalst W M P, Basten T. Inheritance of workflows: An approach to tackling problems related to change. Theoretical Computer Science, 2002, 270(1-2): 125-203

[15] Aalst W M P, Weske M, Wirtz G. Advanced topics in workflow management: Issues, requirements and solutions. Journal of Integrated Design and Process Science, 2003, 7(3): 49-77

[16] Agostini A, DeMichelis G. Improving flexibility of workflow management systems. Business Process Management, Models, Techniques and Empirical Studies, LNCS 1806, 2000: 218-234

[17] Ellis C A, Keddara K, Rozenberg G. Dynamic change within workflow systems//Proceedings of Conference on Organizational Computing Systems (COOCS′95). Milpitas, USA, 1995: 10-21

[18] Kradolfer M, Geppert A. Dynamic workflow schema evolution based on workflow type versioning and workflow migration//Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems (CoopIS′99). Edinburgh, Scotland, 1999: 104-111

[19] Casati F, Ceri S, Pernici B, Pozzi G. Workflow evolution. Data and Knowledge Engineering, 1998, 24(3): 211-238

[20] Rinderle S, Reichert M, Dadam P. Evaluation of correctness criteria for dynamic workflow changes//Proceedings of the 1st International Conference on Business Process Management (BPM′03). Eindhoven, The Netherlands, 2003: 41-57

QIAO Xiao-Qiang, born in 1977, Ph. D. candidate. His research interests are service-oriented computing, service composition and workflow technologies.

WEI Jun, born in 1970, Ph. D., professor, Ph. D. supervisor. His research interests include service oriented computing, middleware, mobile computing, software engineering.

HUANG Tao, born in 1965, Ph. D., professor, Ph. D. supervisor. His research interests include software engineering, distributed computing and middleware system.

Background

The main research of the group focuses on Service Oriented Architecture (SOA) and Service Oriented Computing (SOC), which includes service composition, service discovery and selection, service quality management and service assessment, etc. This work is supported by the National Natural Science Foundation of China under Grant No. 60673112, the National Basic Research Program (973 Program) of China under Grant No. 2002CB312005, and the National High-Tech Research and Development Program (863 Program) of China under grant No. 2006AA01Z163 and No. 2007AA010301.

In order to ensure the correct service collaboration, the interactions among participating services should be compatible and consistent. Mediation technologies provide effective approaches to bridge service differences for correct service collaboration. However, as for the dynamic nature of service-oriented computing, mediation should be equipped with dynamic adaptation and re-configuration capabilities to avoid the bottle-neck of the dynamic management of collaboration system. This paper proposes a mediation approach for service collaboration that preserves the autonomy of concrete services and improves the flexibility of service collaboration. An adaptation method for proposed mediation structure is also presented to ensure the agility and adaptability of service collaboration based on mediation mechanism.