基于软件管道 Actor 模型的 BPEL 流程转化研究

吴众欣[1],钱德沛[1,2],黄泳翔[1]
(1.西安交通大学 电子与信息工程学院,陕西 西安 710049;
2.北京航空航天大学 计算机学院,北京 100191)

摘 要:在当今流行的SOA(面向服务的架构,Service Oriented Architecture)服务体系中,BPEL(Business Process Execution Language)是现今使用最广泛的业务流程执行语言。BPEL定义了业务流程如何与外部Web服务进行交互的过程。但BPEL面向流程编制与业务逻辑设计,是一种控制流模式下的处理过程。控制流模式下的业务流程常常依赖于控制流程的旁置条件与控制指令间的依赖关系.不能产生较好的并发处理过程。而数据流处理模式天生具有并发特性,将控制流模式的业务过程转化为数据流处理模式的执行过程,可提高业务流程执行过程的并发性。采用软件管道Actor模型对BPEL流程进行转化,给出模型结构,转化架构与转换示例,通过实验证明了这种转化可以有效地提高流程并发执行性能。

关键词:数据流;业务流程;控制流;并发
中图分类号:TP393 **文献标识码**:A **文章编号**:1673-629X(2009)07-0004-06

Research on BPEL Process Conversion Based on Actor Model with Pipeline

WU Zhong-xin[1],QIAN De-pei[1,2],HUANG Yong-xiang[2]
(1.School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an 710049,China;
2.School of Computer Science, Beihang University, Beijing 100191,China)

Abstract:BPEL is the most widely used business process execution language in the SOA system. BPEL defined the interaction between the process and outer service. But, the BPEL which belongs to control flow model is oriented to process orchestration and business logic designing. Business process is dependent on the correlation of the control command and side-by condition under the control flow model, which can not get a higher concurrency of system. Data flow model has built-in concurrency. Converting the control flow process to data flow process, to enhance the performance of business process execution is a preferable method. In this paper, converted the BPEL process to Actor model with software pipeline, an actor pipeline model and architecture of conversion is described and an example is proposed. It is much effective and available to enhance the performance of the process concurrency with our experiments results.

Key words:data flow;business process;control flow;concurrency

0 引 言

在当今流行的SOA(面向服务的架构,Service Oriented Architecture)服务体系中,业务流程执行语言(BPEL,Business Process Execution Language)是使用最广泛的业务流程执行语言。BPEL定义了业务流程如何与外部Web服务进行交互的过程。因为BPEL是一种面向业务逻辑设计与流程编制语言,所以BPEL流程是一种控制流模式下的处理过程。控制流模式下业务流程常常依赖于控制流程的旁置条件与控制指令间的依赖关系,不能产生较好的并发处理过程。而在数据流处理模式下,只要计算单元的输入数据准备好就可以运行计算过程,它具有天生的并发特性。将控制流模式的业务流程转化为数据流处理模式的执行过程,可提高BPEL流程的并发性,加快流程执行过程。

笔者考虑采用控制流的业务流程转换为数据流执行过程方法,将BPEL流程转换为数据流处理模式下的管道Actor模型执行过程,增强流程执行能力。数据流计算模型分别由是IBM公司的Karp和Mille、斯坦福大学的Adams、麻省理工的Rodri gue提出并逐渐完善[1~3]。Seeber和Lindquist在高并行系统的相关逻辑设计中采用了数据流处理方法[4]。Whiting提出了有关数据流模型下可重配置计算的方法[5]。数据流处

收稿日期:2009-04-21;**修回日期**:2009-05-21
基金项目:国家高技术研究发展计划863项目(2009AA01Z144)
作者简介:吴众欣(1974-),男,湖南湘潭人,博士,研究方向为Web服务组合;钱德沛,教授,博士生导师,研究方向为计算机网络新技术、高性能计算机体系结构、网格计算。

理模式下的 Actor 模型最早由 Hewitt 提出,后被 Agha 定义为有少数表达原语组成的简化模型[6]。该模型提供封装与异步消息传递机制在面向对象语言领域有多种实现,1991 年 Erlang 语言实现了 actor 模型[7,8]。Actor 模型的实现经历了简单的线程实现方式到采用事件实现方式,现今随着多核与并发程序设计的发展,又出现基于线程与事件的混合方式来实现 Actor 模型,其中最经典的代表就是 Scala 语言,它是由洛桑瑞士联邦理工学院(EPFL)Martin Odersky 教授于 2003 年创建,现成为了 Actor 模型实现的技术热点[9]。

1　管道 Actor 模型与描述语言

Actor 模型是基于独立 actor 异步消息通讯的并发计算模型。计算过程是通过 actor 之间显式消息通讯完成的,每个 actor 是一个自含的、交互的、相互独立的并行计算最小单位。actor 之间不共享状态,拥有自己的资源,利用该资源处理到来的消息[10]。

1.1　管道 Actor 模型

每个 actor 包含信件队列与 actor 处理机。Actor 可用一个三元组,式(1)来表示。

<Name, MailQueue, Behavior>　(1)

式中:

Name - actor 的名字,用具有唯一性的通信地址(mail address)来表示;

MailQueue - 信件队列也称作消息队列,由等待处理的消息集合组成;

Behavior - actor 的行为,代表 actor 处理消息和执行计算的能力。

传递到 actor 的消息被放入邮件队列的缓存中,消息按照先来先服务的顺序被处理。一个 actor 只能通过向别的 actor 发送消息来改变另一个 actor 的行为。如果 actor a 知道 actor b 的地址,那么 a 是 b 的熟人。

当接到一个消息时,actor 的行为可做出的三种基本操作,生成新的 actor,发送消息和产生替代行为。在 Agha 模型中由 create,send 和 become 原语来表示。

* Create 原语。创建新的 actor,返回新 actor 的地址并传给其他的 actor。create 指定新 actor 的初始行为。

* Send 原语。用来进行消息传递,一个消息包含目的 actor 地址,被调用的方法名和引用该方法所需的参数。将消息发送到接收 actor 的消息队列中,是在 actor 之间进行异步通信的过程。

* Become 原语。指明一个替代行为,是 actor 在处理当前消息结束后的行为定义。

如图 1 所示,当 actor 处理机 X_n 接收到信件队列中第 n 个消息时,它将创建一个新的 actor 处理机 X_{n+1} 来执行替代行为,新的 actor 处理机将指向信件队列的 $n+1$ 消息位置。X_n 和 X_{n+1} 不会相互影响对方的行为,确切地说 X_n 只处理第 n 个消息。X_n 还可以创建新的 actor Y_1(注意:X_n 与 Y_1 是不同的 actor,而 X_{n+1} 与 X_n 是相同的 actor)。X_n 也可以向 actor Z_m 的消息队列发送消息。

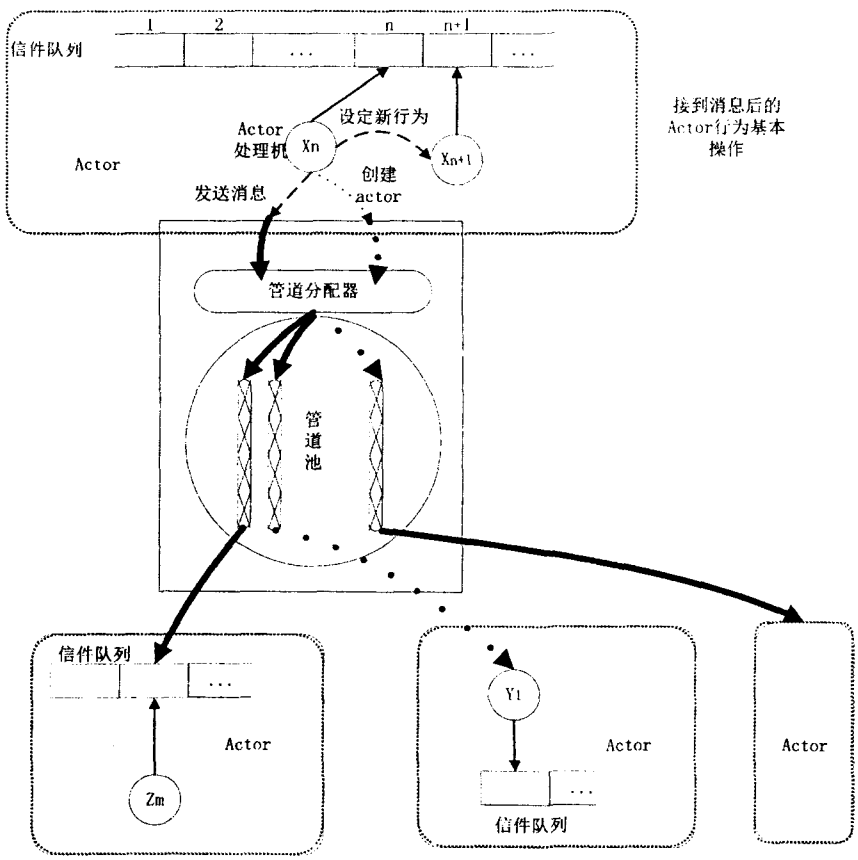


图 1　软件管道 Actor 模型

软件管道是一种为面向服务的业务程序提供吞吐率控制,负载均衡的软件架构。基于软件管道,可对消息传递过程进行流量控制。为 actor 模型添加软件管道,使得 actor 之间通信获得流控能力,如图 1 所示[11]。管道架构是一种 plug - in 模式,actor 之间的通信显式地使用管道,而管道的分配控制过程对于 actor 实例而言是不可见的。

1.2　简单 Actor 语言

简单 Actor 语言(SAL, Simple Actor Language)包含一系列的行为定义与指令(command),如式(2)所

示。

$$Pgm::= BDef_1 \cdots BDef_n Com \quad (2)$$

式中：

BDef – actor 的行为定义，它是 actor 的行为模板；

Com – 发给 actor 的指令。

SAL 语法组成部分如下：

1) 表达式。

设定表达式为 e，actor 的名字为 u, v, w, x, y, z。

2) 指令。

指令语法如式(3)所示：

$$\begin{aligned}
&Com::= \text{send}[e_1, \cdots, e_n] \text{ to } x \quad \text{(message send)}\\
&\quad \text{become } B(e_1, \cdots, e_n) \quad \text{(new behavior)}\\
&\quad \text{let } x_1 = [recep]\text{new } B_1(e_1, \cdots, e_{i1}) \quad (3)\\
&\quad \cdots x_k = [recep]\text{new } B_k(e_1, \cdots, e_{ik})\\
&\quad \text{in Com} \quad \text{(actor creations)}\\
&\quad \text{if e then } Com_1 \text{ else } Com_2 \quad \text{(conditional)}\\
&\quad \text{case x of } (y_1 : Com_1, \cdots, y_n : Com_n) \quad \text{(name matching)}\\
&\quad Com_1 \,||\, Com_2 \quad \text{(composition)}
\end{aligned}$$

式中：

message send – 消息发送过程，表达式 e_1 到 e_n 计算的结果被发送到 actor x。执行指令的过程不包含实际传送消息的过程；

new behavior – 指定 actor 的新行为。新的行为标识为 B。表达式 e_1 到 e_n 的计算结果绑定于 B 的熟人列表(acquaintance list) 参数；

actor creations – 创建具有一定行为的 actor。recep 是可选标识符，该标识符对应接待 actor (receptionist actor)，它负责接收消息；

conditional – 条件执行过程，表达式 e 的结果是布尔类型，如果结果为 true，指令 Com_1 执行，否则 Com_2 执行；

name matching – 名字匹配过程，名字 x 与名字集合 $y_1, \cdots, y_n$ 匹配，如果存在匹配的名字，执行相应的指令；

composition – 指令并发执行过程。

3) 行为定义。

行为定义的语法如式(4)所示：

$$\begin{aligned}
&BDef::= \text{def} < \text{beh name} > (< \text{acquaintence list} >)[< \text{input list} >]\\
&\qquad Com \quad (4)\\
&\text{end def}
\end{aligned}$$

式中：

〈beh name〉– 行为名字；

input list – 行为参数；

acquaintence list – 熟人列表；

Com – 行为执行体。

1.3 转换过程与主要实现类

将 BPEL 流程转化为软件管道 actor 模型处理过程是将 BPEL 过程中的各个活动转化为 actor 实例，活动之间的数值传递通过 actor 实例以消息方式进行分发，如果活动之间没有数据关联，那么这些 actor 之间的运行就是多个 actor 实例通过软件管道与外部服务交互、协同工作的过程。

Actor 模型的实现在技术上经历过基于线程、基于事件和基于二者混合实现的三个阶段，文中也是以线程与事件的混合方式来实现 Actor 模型[12,13]。转化过程的主要实现类如图 2 所示。

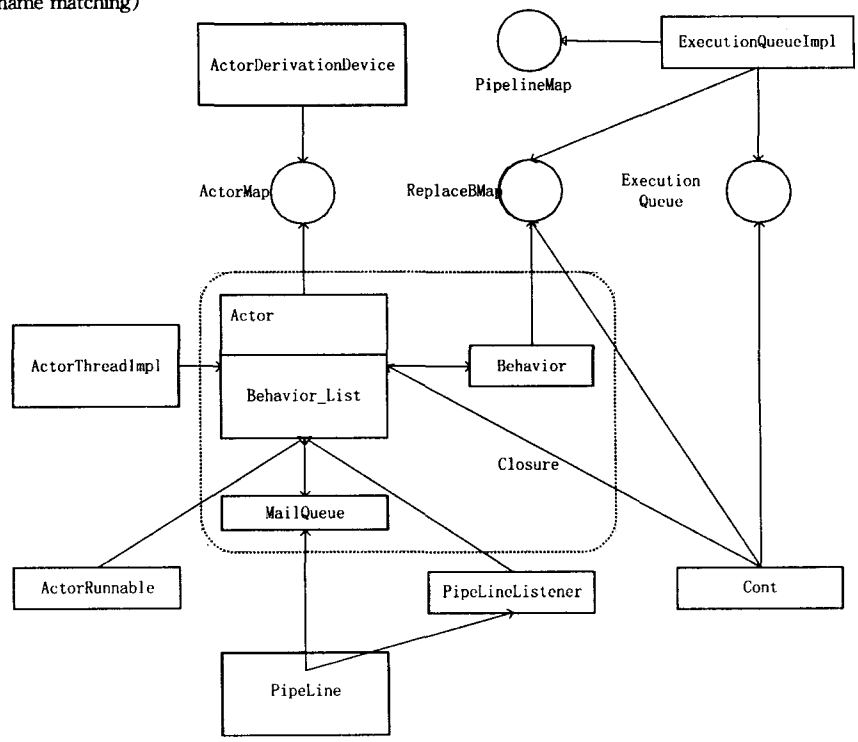


图 2 转化过程的主要实现类图

1) Actor 派生器(ActorDerivationDevice)。

实例化的 Actor 要在 Actor 派生器中注册。执行时需注入到派生器的上下文中执行。

2) Actor 类组。

Actor 是一个闭包，闭包是由函数及其相关的引用环境组合而成的实体。除了自身的 actor 类之外，它还包含信件队列类(可被管道类调用)与行为类(使用替代行为 Map 接口)。

3) Cont 类。

该类是 actor 的后继(continuation)类，通过闭包机

制,它包含了 actor 类的动作列表,同时它是执行队列的使用者。

4) 管道(Pipeline)类。

管道是 actor 与服务两两之间通信的通道,它在调用与执行过程中起到解耦和作用,同时也提供了流量控制作用。由于文中主要讨论的是 BPEL 的转化过程,所以对于管道的流控实施过程不做过多的描述。

5) 执行队列(ExecutionQueueImpl)类。

队列保存活动与管道的引用,由后继 actor 来执行。

2 转换过程示例

使用 BPEL 编制的阶乘流程文件如下:

```
<process name="factorialService"
    ...
    <sequence>
    <receive partnerLink="receiver"operation="factorial"
    variable="input"createInstance="yes"/>
    <if>
    <condition> $ input.value & gt; =1</condition>
    <sequence>
    <assign>
    <copy>
    <from> $ input.value-1</from>
    <to variable="callIn"part="value"/>
    </copy>
    </assign>
    <invoke name="recursive Call"partnerLink="invoker"
    operation="factorial"inputVariable="callIn"
    outputVariable="callOut"/>
    <assign>
    <copy>
    <from> $ input.value * $ callOut.value</from>
    <to variable="output"part="value"/>
    </copy>
    </assign>
    </sequence>
    <else>
    <assign>
    <copy>
    <from>1</from>
    <to variable="output"part="value"/>
    </copy>
    </assign>
    </else>
    </if>
    <reply partnerLink="receiver"operation="factorial"
    variable="output"/>
    </sequence>
    </process>
```

从 BPEL 阶乘流程文件的 receive 语句开始,创建一个 PROCESS 实例,该实例是所有活动的上下文环境。在派生器中注册一个 actor a,创建 actor 对象,将其行为封装为闭包,接收消息,替换行为,创建管道,按照对活动的调用次序,创建后续 actor,同时也创建后续 actor 的管道,将活动与管道的引用加入到执行队列中。将 actor a 注入派生器中,派生器遍历 a 的执行动作队列,发送、接收消息,同时执行后续活动的注册过程。

转换过程伪代码如下:

```
PROCESS I=new PROCESS();
    ActorDerivationDevice,register(i){
        c=new actor(behaviorList);
        c.receive();
        eq=new ExecutionQueueImpl();
        p=new pipeline();
        for(numOfActivity){
          continuationActor cont=new actor(Activity a);
          enqueue(Activity,eq);
          }
        ActorDerivationDevice.inject(c);
    }
    ActorDerivationDevice.inject(actor k){
    for(number OfContinuationActor){
    dequeue(Activity,eq);
    pineline p=new pineline(a);
    p.send();
    p.receive();
    ActorDerivationDevice.register(cont);
    }
}
```

转换为 actor 模型后,actor 实例执行过程的伪代码如下:

```
    def Factorial()[value,a]
    Become Factorial()||
    if value=0
    then send[1]to a
    else let cont=new FactorialCont(value,a)
    in send[value-1;cont]to self
    end def
    def FactorialCont(Value,a)[arg]
    send[value□
    end def
```

首先将整数 n 与 actor 名字 a 赋给阶乘 actor,接到消息之后 a 创建后续(continuation)actor cont 并向 cont 发送一个内容为 $n-1$ 的消息。每当接收到一个整数,

后续 actor 的行为是将这个整数与自身持有的变量相乘并将结果返回。使用 Actor 模型计算阶乘 3 的过程如图 3 所示，图中的竖线表示 actor 的时间线，时间线的顶部箭头表示创建了一个 actor，其他线表示为消息。计算结果返回给 actor a。因为计算阶乘的 actor 是无状态的，所以能同时处理不同的请求，并发执行，而不影响阶乘的计算结果。

如图 4 所示，如果阶乘的每次相乘流程活动次序为图 4(a)，若采用 BPEL 流程执行方式则是各个相乘流程占用一个线程，两个流程并行执行过程如图 4(b)。而在带有软件管道的 Actor 模型下两个相乘流程是并发执行，如图 4(c)所示，各个活动的次序会内部流程有序，但整体执行无序的状态下并发执行。所以业务流执行过程的性能得到提升。

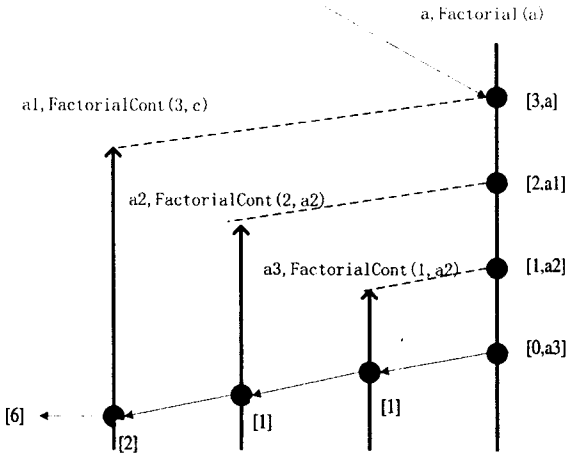
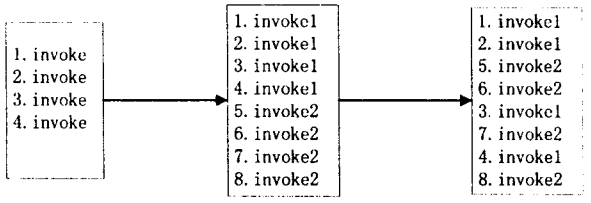


图 3 阶乘 3 Actor 模型计算过程示例



(a) 单一阶乘3流程　(b) 两个阶乘3流程并行执行过程　(c) 两个阶乘3流程在actor模型下并发执行过程

图 4 流程并发过程

3 实 验

文中实验环境如下：流程服务器配置为 Dual Xeon 2.8GHz CPU，2.0G 的内存，Windows XP Professional 操作系统，JDK1.5，SOAPUI 1.6 load - testing tool。MapReduce 数据处理服务器是 4 台 Pentium Dual Core 1.7GHz CPU，4G 内存，Windows Vista Basic 操作系统，JDK5.0，Hadoop0.18。业务流程为 hadoop 平台上进行标引过程中位置信息的 MapReduce 操作。

使用 SOAPUI 测试集触发数据 MapReduce 服务，得到实验结果如图 5 所示，由于 MapReduce 过程中消息交互频繁(需要交互 Map 输入与输出的消息与 Reduce 计算的输入与输出的消息)，管道起到的流控作用明显，基于软件管道的 actor 模型执行性能提升很突出。BPEL 执行过程在线程的并行执行期间，需要一定的同步过程，即 Reduce 过程等待 Map 的数值结果来继续计算，所以在大数据量情况下，BPEL 的解释执行过程比 Actor 模型过程的流程执行性能要低。而在 Actor 模型中，Map 的结果数据一旦生成，Reduce 即可执行，所以虽然有一定的 actor 对象内存的消耗，但并发处理过程会带来整体执行性能上的提升。在具体实现过程中 Actor 处理机对象，要么马上销毁，要么重复利用，Actor 模型转换过程中对内存的占用尤其值得关注。

使用 sieve 算法筛选 100 以内的质数的业务执行时间实验，如图 6 所示。本实验环境只是在流程服务器上运行，由于 actor 的个数不多，软件管道的通信控制能力没有得到充分的体现。但在 actor 数量对任务处理的性能提升方面，该实验可以很好地展示。随着 actor 实例创建的数量增大，并发处理能力的提高，执行时间变短，整体业务流程的执行能力获得提升。

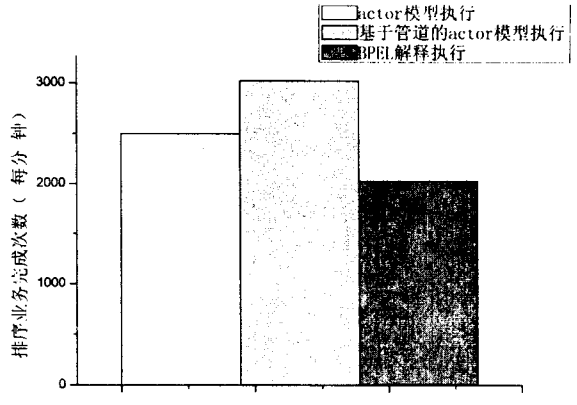


图 5 MapReduce 业务流程完成次数

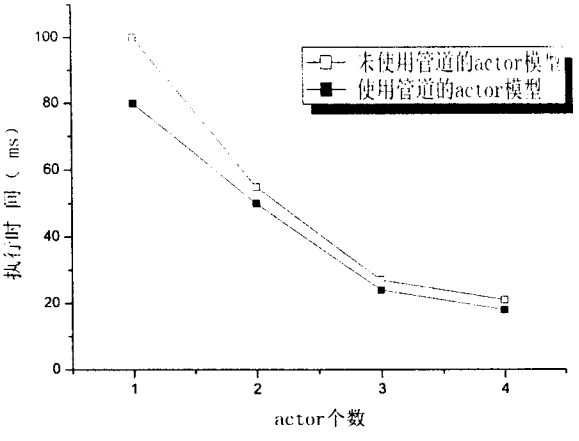


图 6 actor 实例个数对执行时间的影响

4 结束语

采用带有软件管道的 Actor 模型对 BPEL 执行流程进行转化,给出模型结构、流程转化过程、转化过程的主要实现类,与转化示例。通过实验验证了将 BPEL 业务流程转化为软件管道的 Actor 模型处理过程,可使得流程执行的并发性得到了提升,随着 actor 实例数量的增加,降低了流程的执行时间,提高了流程执行性能。

参考文献:

[1] Karp R M, Miller R E. Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing [J]. SIAM J. Applied Mathematics, 1966, 14(11): 1390 - 1411.

[2] Adams D A. A Computation Model with Data - Flow Sequencing[R]. Stanford, Calif: Computer Science Dept., Stanford Univ, 1968.

[3] Rodriquez J E. Graph Model for Parallel Computation[R]. Cambridge, Mass: MIT, 1967.

[4] Seeber R R, Lindquist A B. Associative Logic for Highly Parallel Systems[C]//Proceedings of AFIPS Joint Computer Conferences. New York, NY, USA: ACM, 1963: 489 - 493.

[5] Whiting P G, Pascoe R S V. A History of Data - Flow Languages[J]. IEEE Annals of the History of Computing, 1994, 16(4): 38 - 59.

[6] Agha G A. ACTORS: A Model of Concurrent Computation in Distributed Systems[M]. Cambridge, Massachusetts: MIT Press, 1986.

[7] Armstrong J. The development of Erlang[C]//Proceedings of the second ACM SIGPLAN international conference on Functional programming. New York, NY, USA: ACM, 1997: 196 - 203.

[8] Scalas A, Casu G, Pili P. High - performance technical computing with erlang[C]//Proceedings of the 7th ACM SIGPLAN workshop on ERLANG. New York, NY, USA: ACM, 2008: 49 - 60.

[9] Haller P, Odersky M. Scala Actors: Unifying thread - based and event - based programming[J]. Theoretical Computer Science, 2009, 410(2 - 3): 202 - 230.

[10] Clinger W D. Foundation of Actor Semantics[D]. Massachusets: Massachusets Institute of Technology, 1981.

[11] Douillet A, Gao Guang R. Software - Pipelining on Multi - Core Architectures[C]//Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. Washington, DC, USA: IEEE Computer Society, 2007: 39 - 48.

[12] Haller P. An object - oriented programming model for event - based actors[D]. Fakultt fr Informatik, Universitt Karlsruhe, 2006.

[13] Van Cutsem T, Mostinckx S, De Meult W. Linguistic symbiosis between event loop actors and threads[J]. Computer Languages, Systems and Structures, 2009, 35(1): 80 - 89.

(上接第 3 页)

在时域还是频域,小波分解与傅里叶变换耦合法的处理效果都很好,适合处理静电监测信号,证明了该方法的有效性。

4 结束语

针对静电监测信号信噪比极低、工频干扰大、故障信号难以提取等特点,根据傅里叶变换和小波阈值降噪理论,提出小波分解与傅里叶变换耦合法,与常规滤波法对比后得出结论:小波分解与傅里叶变换耦合法滤波效果最明显,均方根误差最小,信噪比是常规滤波方法的 2~3 倍,表明无论是对于仿真信号还是实验信号,小波分解与傅里叶变换耦合法不仅具有很强的噪声抑止能力,同时能很好地提取出异常信号,适合用来做静电感应信号的滤波处理。

参考文献:

[1] Powrie H E G, McNicholas K. Gas Path Condition Monitoring During Accelerated Mission testing of a demonstrator engine [C]//AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit, 33rd. Seattle, WA: [s. n.], 1997: 6 - 9.

[2] Powrie H E G, Fisher C E. Engine health monitoring: towards total prognostics[C]//Snowmass at Aspen, CO, USA: IEEE Aerospace Applications Conference Proceedings. US: [s. n.], 1999: 11 - 20.

[3] Fisher C E. Gas path debris monitoring - a 21st century PHM tool[C]//IEEE Aerospace Conference Proceedings. Montana: [s. n.], 2000: 441 - 448.

[4] Sorokin A, Arnold F. Electrically charged small soot particles in the exhaust of an aircraft gas - turbine engine combustor: comparison of model and experiment[J]. Atmospheric Environment, 2004, 38: 2611 - 2618.

[5] 吴　婷,颜国正,杨帮华. 基于陷波器和小波变换去除自发脑电信号噪声的方法[J]. 测试技术, 2007, 26(4): 29 - 31.

[6] 胡昌华,李国华,刘　涛,等. 基于 MATLAB 6.X 的系统分析与设计—小波分析[M]. 西安: 西安电子科技大学出版社, 2004: 292 - 296.

[7] 吴振磊,陈钟荣. 基于小波变换的信号突变检测[J]. 微计算机信息, 2006, 34: 747 - 758.

[8] 朱　华,吴传生,汪小梅. 一种改进的小波消噪阈值选取方法[J]. 计算机应用, 2007, 27(10): 2605 - 2609.