ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.17, No.6, June 2006, pp.1349−1359
DOI: 10.1360/jos171349

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种形式化的动态体系结构描述语言*

李长云[1,2+], 李赣生[1], 何频捷[1]

[1](浙江大学 计算机软件研究所,浙江 杭州 310027)
[2](湖南工业大学 计算机系,湖南 株洲 412008)

A Formal Dynamic Architecture Description Language

LI Chang-Yun[1,2+], LI Gan-Sheng[1], HE Pin-Jie[1]

[1](Institute of Computer Software, Zhejiang University, Hangzhou 310027, China)
[2](Department of Computer, Hu'nan University of Technology, Zhuzhou 412008, China)
+ Corresponding author: Phn: +86-733-2622469, Fax: +86-733-2622985, E-mail: lcy469@163.com

Li CY, Li GS, He PJ. A formal dynamic architecture description language. *Journal of Software*, 2006,17(6): 1349−1359. http://www.jos.org.cn/1000-9825/17/1349.htm

Abstract: A dynamic architecture description must provide a dynamic behavior specification of software system. Based on the high-order multi-type π calculus theory, a dynamic architecture description language D-ADL is proposed. In D-ADL, components, connectors, and architecture are modeled as the 'Abstraction' type of high-order π calculus; system behavior is modeled as the 'Process'; and the interaction point between component and connector is modeled as the 'Channel'. D-ADL separates the dynamic behavior expressed in an explicit way from the computation behavior, which is helpful to edit and understand the changing logic in system. The result of the dynamic behavior can be beforehand deduced because this behavior is formalized as the high-order 'Process'. On the basis of D-ADL specification, rules about applying the theory of behavior equivalence and behavior simulation in π calculus to online evolution and architecture refinement are proposed. How D-ADL can be used is illustrated through a case study.

Key words: software architecture; ADL; component; high-order multi-type π calculus; dynamic behavior

摘 要: 描述动态体系结构的关键在于如何刻画软件的动态行为.基于高阶多型π演算理论,提出了动态体系结构描述语言 D-ADL(dynamic architecture description language).在 D-ADL 中,构件、连接件和体系结构风格被模型化为高阶多型π演算中的抽象(abstraction)类型,系统行为被模型化为进程(process),构件和连接件的交互点则被模型化为通道(channel).为方便系统变更逻辑的编写、修改和理解,D-ADL 将动态行为从计算行为中分离出来,显式、集中地表达.由于动态行为可形式化为高阶π演算进程,其结果因此能够被预先推导.在 D-ADL 规约框架下,给出了将π演算的行为模拟和等价理论用于系统联机演化和体系结构模型求精的规则.实际案例说明了 D-ADL 的应用.

* Supported by the National Natural Science Foundation of China under Grant No.60373062 (国家自然科学基金); the Natural Science Foundation of Hu'nan Province of China under Grant No.04JJ3052 (湖南省自然科学基金); the Fork Ying Yung Science Foundation for Yong Teacher under Grant No.94030 (霍英东青年教师基金)

Received 2006-01-07; Accepted 2006-03-28

关键词: 软件体系结构;ADL;构件;高阶多型π演算;动态行为
中图法分类号: TP311 文献标识码: A

近年来,Internet 已成为主流的软件运行环境,网络的开放性和动态性使得研究者日益关注具有自适应能力软件的开发.自适应软件的一个基本特征是能够在运行时进行演化,以适应需求和环境的变化[1].软件体系结构(software architecture,简称 SA)从全局的角度为系统提供结构、行为和属性等信息,已经成为软件开发过程中的核心制品[2].体系结构描述语言(architecture description language,简称 ADL)是 SA 研究的核心问题[3].如何在体系结构层次上刻画软件的运行时动态行为,如何描述动态的 SA,即动态 ADL 的研究已成为设计和实现自适应性软件的基础和关键.

Oquendo[4]等人认为,SA 至少应该从两个不同的视图描述:结构视图和行为视图.结构视图着眼于系统的组成元素及其互连拓扑结构;行为视图着眼于系统的功能和行为.有两种不同类型的行为应该加以区分:计算行为和动态行为.计算行为面向系统的商业逻辑,处理业务功能中的数据信息;而动态行为面向系统的预定义演化逻辑,使系统能够自适应演化,以体系结构元素为处理对象,如增删构件、建立新的连接等.静态 ADL 如 Wright[5]和 Darwin[6]等关注系统的结构视图和计算行为视图,忽略系统的动态行为.而具备对动态行为的表示能力是动态 ADL 的显著标识.因此,设计动态 ADL 的关键之一在于采取什么样的方法表示动态行为.对于动态行为的表示机制,有两种设计思路:一是增强计算行为的表达能力,使之容纳演化逻辑的表示;另一种是将动态演化逻辑与计算行为相分离,显式地、集中地表达.后者比前者更优越,符合软件工程中关注分离的原则,有助于系统的动态调整.

另一方面,为便于 SA 模型的验证、求精和实现,SA 规约应该具有精确的语义.本文提出的 D-ADL(dynamic ADL)是一个形式化的基于高阶多型π演算的动态 ADL.D-ADL 显式、独立地表示动态行为,并将高阶多型π演算作为计算行为和动态行为的统一语义基础.借助高阶多型π演算理论,D-ADL 在设计阶段可以精确地刻画系统的交互行为,便于模型验证和求精;在系统投入运行时,有利于推导系统的行为,支持系统的在线演化.

本文首先简单介绍高阶多型π演算概念;然后介绍 D-ADL 语言的描述框架和类型系统,讨论 D-ADL 语法规约及其语义,重点阐述 D-ADL 对行为尤其是动态行为的表示及其形式化解释;接着,使用π演算理论探讨 D-ADL 对系统联机演化和 SA 求精的支持;最后是相关工作的分析、结论与展望.

1 高阶多型π演算简介

π演算是 20 世纪 90 年代计算机并行理论领域最重要的并发计算模型,它由 Milner 等人[7]对 CCS(calculus of communicating systems)进行扩充而得到.高阶多型π演算由 Sangiorgi[8]在一阶π演算的基础上发展,旨在描述结构和行为都不断变化的并发系统.π演算具有 3 个基本的实体:进程(process)、名(name)和抽象(abstraction).进程是并发运行实体的单位,并以名来统一定义通道(channel)以及在通道中传送的对象(object),每个进程都有若干与其他进程联系的通道,进程通过它们共享的通道进行交互.高阶多型π演算和一阶π演算关键的不同在于对象名本身也可以是进程,这就使得交互中通信的数据可以是进程,进程可以被传递.抽象建立在进程的基础上,是带参数的进程.对抽象的参数进行具体化,得到一般的进程.在高阶多型π演算中,参数以及通道中传送的对象都具有类型,对抽象的参数进行具体化以及进程交互时要注意类型兼容.

定义 **1**. 设 N 表示名(name)的集合,名由小写字母表示, $\overline{N}\triangleq(\overline{\alpha}\mid\alpha\in N)$, $\alpha\in N\cup\overline{N}$,P,Q 表示高阶π演算进程,则高阶多型π演算进程可定义为

$$P ::= 0 \mid \alpha x.P \mid \overline{\alpha} y.P \mid P+Q \mid P \mid Q \mid \nu x P \mid [x=y]P \mid D\langle \tilde{K} \rangle,$$

其中 $D::=(\tilde{x})P$.其意义解释如下:

0 表示非活动进程,它不能与任何进程交互;

$\alpha x.P$:这里,α是指某一通道的输入端,此进程的行为是先通过通道名α输入接收对象 y,再激活进程 $P[y/x]$,其中 x 为局部名(local name),$[y/x]$表示 a 换名操作;

$\overline{\alpha}\,y.P$:这里,$\overline{\alpha}$ 是指某一通道的输出端,此进程的行为是先通过通道名α输出对象名 y,再激活进程 P,其中 y 为全域名(global name);

$P+Q$:这是进程的不确定计算形式,此进程将根据一定的诱因来激活进程 P 或 Q,它的行为仅是这个被激活进程的行为;

$P|Q$:这是一个并行操作形式的进程,此式中进程 P 和 Q 并行存在,它们可以独立地与其他进程进行交互操作,也可以彼此通过共享的通道进行通信;

$vx.P$:该进程中,x 是进程 P 的局域名,它使得进程 P 不能通过通道 x 与其他进程进行通信;

$[x=y]P$:这是一个条件进程,如果 $x=y$,它将激活进程 P,否则将不进行任何操作,成为非活动进程 0;

D 是抽象,定义为$(\tilde{x})P$,表示以 $\tilde{x}$ ($\tilde{x}$ 代表 $x_1,x_2,\ldots,x_n$)为参数的进程.$D\langle\tilde{K}\rangle$表示对 D 进行具体化,值 $\tilde{K}$ ($\tilde{K}$ 代表 $K_1,K_2,\ldots,K_n$)对参数 $\tilde{x}$ 赋值.其中 $\tilde{x}$ 和 $\tilde{K}$ 类型兼容.

高阶多型π演算通过归约(reduction)规则和迁移(transition)规则严格定义进程的演化,归约规则和迁移规则参见文献[8].

2 D-ADL 的语法规约和形式语义

本节讨论 D-ADL 对 SA 的描述方法,重点阐述 D-ADL 对行为的表示及其形式化语义解释.D-ADL 语法使用扩充的 BNF 范式表示,全部语法规范可参见文献[9].

2.1 D-ADL的描述框架和类型系统

一方面,D-ADL 遵循 Wright[5]等给出的已被广泛认同的 SA 描述框架,围绕 SA 实体如构件、连接件和配置等进行 SA 建模,如图 1 所示;另一方面,D-ADL 将高阶多型π演算作为行为语义基础.凭借高阶π演算描述动态系统的特征,D-ADL 允许构件、连接件和配置产生变更,并使得对 SA 的自动化分析成为可能.

构件(component)是 SA 的基本要素之一,依据接口和行为描述.在 SA 中,构件被视为用于实现业务逻辑的实体,执行系统的计算功能,此外,构件还描述系统的动态行为.构件作为一个封装的实体,仅通过其接口与外部环境交互,而构件的接口由一组端口(port)组成.每个端口由一组通道(channel)组成,因此端口可看成通道集合.引入端口让构件的交互功能结构化、局部化,从而方便对其进行分析.通道是最基本的交互点,它的角色就是在 SA 元素之间提供通信渠道,构件通过通道发送或接收值.

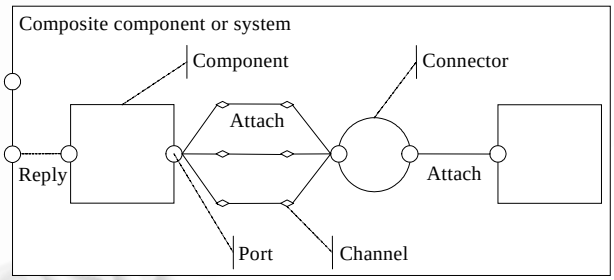

Fig.1 The description framework of D-ADL

图 1 D-ADL 描述框架

连接件(connector)是一种特殊的构件,旨在建立构件间的交互以及支配这些交互规则.连接件依据接口和路由行为(routing)描述.显然,连接件的接口也由一组端口构成,连接件的端口同样由一组通道组成.路由行为是计算行为的一个变种,指示构件间的通信轨迹和执行交互规则及约束.

复合构件(复合连接件)由外部端口和内部元素构成.应用系统本身可被视作一个特殊的复合构件.复合构件(复合连接件)的内部元素由一系列成员构件和成员连接件组成,它们通过配置链接在一起.配置要求构件端口(通道)与连接件端口(通道)显式连接.一个构件不能直接连接到另一个构件,构件之间通信和协作只能通过连接件进行.

D-ADL 将高阶多型π演算作为行为语义的基础.D-ADL 拥有丰富的数据类型,总体上分为两种:常规类型(OrdinaryType)和体系类型(ArchitectureType).常规类型主要容纳一般意义的数,如整数、字符串和布尔值等;体系类型用于描述体系结构元素,如构件、连接件等.D-ADL 的基本类型和高阶多型π演算间的对应关系见表 1.

体系类型中最重要的是进程类型(ProcessType),对应高阶π演算中的进程元素,表示体系结构中的各种行为

和动作序列.构件的计算行为、连接件的路由行为以及体系结构的动态行为都属于进程类型,分别对应 computation,routing 和 choreographer 类型.计算行为处理的是系统业务功能逻辑中的数据信息,只能是常规类型值才能够被传递、处理,因此仅被形式化为一阶进程;动态行为面向的主要就是体系结构本身,以体系结构元素为处理对象,使系统能够动态演化,而连接件的路由行为也可以使用所有类型值作为参数传递对象,因此都被形式化为高阶进程.高阶多型π演算中的 Abstraction 概念被 D-ADL 直接使用.Abstraction 是对行为的抽象,用来表示构件类型、连接件类型和体系结构风格:当 Abstraction 是对 Process 的直接参数化抽象时,Abstraction 表示构件类型和连接件类型;当 Abstraction 是对另一个 Abstraction 的参数化抽象时,Abstraction 表示体系结构风格.

Table 1 The mapping relationship between base type in D-ADL and high-order multi-type π calculus

表1 D-ADL的基本类型和高阶多型π演算间的对应关系

<table>
<tr><td colspan="3">D-ADL tag</td><td colspan="2">Interpretation of high-order π calculus</td><td>Remark</td></tr>
<tr><td colspan="3">Type</td><td colspan="2">Sort</td><td>Type of element</td></tr>
<tr><td colspan="3">OrdinaryType</td><td colspan="2">All name except channels in first-order process</td><td>Ordinary value type</td></tr>
<tr><td rowspan="4">Architecture-Type</td><td colspan="2">Channel type</td><td colspan="2">All channel</td><td>Channel type</td></tr>
<tr><td rowspan="3">Process-Type</td><td>Computation</td><td rowspan="3">Process sort</td><td>First-Order process</td><td>Computation of component business</td></tr>
<tr><td>Routing</td><td>High-Order process</td><td>Route behavior of connector</td></tr>
<tr><td>Choreographer</td><td>High-Order process</td><td>Dynamic behavior of architecture</td></tr>
<tr><td colspan="2">Component, connector, style</td><td colspan="2">Abstraction</td><td>Architecture style, component type and connector type</td></tr>
</table>

2.2 构件规约及其计算行为语义

D-ADL 将类型与实例区分开来.构件类型(本文有时直接称其为构件)是实现构件重用的手段.构件具有 3 个基本组成部分:接口部分、行为部分和属性部分.鉴于本文重点在于讨论 D-ADL 的行为语义,关于构件接口和属性的 D-ADL 表示可参见文献[9].D-ADL 将构件行为分为两类:计算行为和动态行为.本节主要讨论计算行为规约(computation),动态行为规约(choreographer)见第 2.3 节.

构件分两种:原子构件和复合构件.原子构件是指不具备内部结构的构件,语法如下:

Atomiccomponent::=**atomiccomponent** *name*(*name*$_1$:*Type*$_1$,…,*name*$_n$:*Type*$_n$)
{[0+*Type definition*.] [0+*Variant definition*.] [01 **port** {[0+Port definition.]}]
computation {*computation*}}[01 **choreographer** {choreographer}]
[01 **property** {property}]}

构件可以参数化,进一步促进重用.Computation 部分描述构件的业务计算逻辑,执行各个端口规约的交互行为和内部计算活动,是原子构件计算功能的完整规约.Computation 语法定义如下:

computation::=*prefix*. *computation*|**if** *boolean* **then** {*computation*$_1$} [01 **else** {*computation*$_2$}]|
choose {*computation*$_1$,…,*computation*$_{n-1}$ **or** *computation*$_n$}|**inaction**|**replicate**|
unobservable. *computation*|*OrdinaryType variant assignment*. *computation*
prefix::=**via** *port*∧*channel* **send** *OrdinaryTypeValue*|**via** *port*∧*channel* **receive**
OrdinaryTypeVariable

prefix 是输入输出动作,via…receive…通过端口通道接收普通类型数据,via…send…通过端口通道发送普通类型数据;if…then…是条件选择语句;choose…or…是随机选择语句;inaction 是哑活动,可作为进程的结束符或 choose 子句的空选择;replicate 是计算的复制或重复;unobservable 是不可观察内部行为;assignment 是变量赋值语句,语法规格与一般程序语言相同.

为了支持系统的层次化分解,引入复合构件的概念.复合构件由多个构件实例和连接件实例组装而成,在规约层次上表达了成员之间的组合.下面是复合构件语法规格:

compositecomponent::=**compositecomponent** *name*(*name*$_1$:*Type*$_1$,…,*name*$_n$:*Type*$_n$)
{[0+type definition.][0+variant definition.][01 **port** {[0+port.]}] [0+Variant assignment.]

instance {[0+Instance.]}

configuration {[01 configuration]}[01 **choreographer** {choreographer}]

[01 **property** {property}]}

复合构件的参数声明、类型和变量定义都与原子构件相同.instance 段声明成员元素(含构件实例和连接件实例),configuration 表示复合构件内的拓扑关系.Configuration 的语法规格如下:

configuration::=rely.configuration|attach.configuration|**inaction**

rely::=*ComponenInstance*∧*Port* **rely** *Port*|*ComponenInstance*∧*Port*∧*Channel* **rely** P*ort*∧*Channel*

attach::=**attach** *ComponenInstance*∧*Port* **to** C*onnectorInstance*∧*Eport*|

attach *ComponenInstance*∧*Port*∧*Channel* **to** C*onnectoInstance*∧*Eport*∧*Echannel*

rely 将复合构件外部端口(通道)映射到成员构件端口(通道)上,attach 将成员构件端口(通道)与成员连接件端口(通道)连接起来.

上述构件计算行为语法规约的π演算语义解释见表 2.

Table 2 The π-calculus semantic interpretation for the specification of component computation behavior

表2 构件计算行为规约的π演算语义解释

D-ADL tag		Interpretation of π calculus		Remark
Computation	if…then…	First-Order process	$[x=y]P$	Condition selection
	choose…or…		$P+Q$	Indefinite selection
	Unobservable		τ	Internal unobservable behavior
	Inaction		0	Inaction
	Replicate		!	Replication or repetition of computation
Prefix	via…receive…	Action	αx	Receive ordinary type data via port channel
	via…send…		$\overline{\alpha}y$	Send ordinary type data via port channel
Configuration	…reply…	α operation		Channel associated by rely or attach is renamed the same name by α operation
	Attach..to…			

构件的计算行为(computation)和动态行为(choreographer)的关系为选择组合.当将构件行为解释为π演算进程时,设若构件行为用进程 P 表示,计算行为用进程 P_a 表示,动态行为用进程 P_b 表示,则 $P=P_a+P_b$.另外,复合构件处理业务逻辑的计算行为来自其成员行为的并发组合,其执行也被委托给成员构件和成员连接件.设若复合构件计算行为用进程 P_a 表示,成员行为分别为进程 $P_{a1},P_{a2},\ldots,P_{an}$,则 $P_a=P_{a1}|P_{a2}|\ldots|P_{an}$.

连接件是一种特殊的构件,同样地,连接件也分为原子连接件和复合连接件两类.原子连接件语法规约类似于原子构件,仅仅是计算行为(computation)描述换成了路由行为(routing,计算行为的一个变种).通过结构化组合,多个连接件和构件也能形成复合连接件.关于连接件的语法规约请参见文献[9].

我们以一个商品订购系统为例来说明 D-ADL 的使用.因为重点是讨论如何使用 D-ADL 进行 SA 建模,所以对商品订购系统适当作了一些简化,省略了细节问题.其体系结构如图 2 所示.

工作流程是,客户首先向商家发出货物订购(order)请求,商家根据情况确定是否能够满足客户的订购请求,向客户发出响应(response).响应信息包含是否同意请求,可能还包含货款信息;若商家同意订购请求,客户支付货款并向商家发出款已付信息(pay),商家确认已付帐(confirm)并发货,订购完成.D-ADL 描述如下:

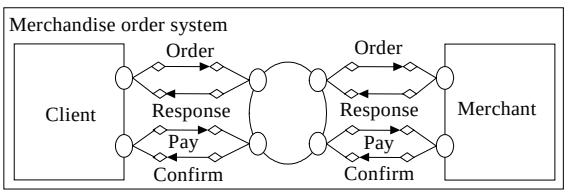

Fig.2 The architecture of commodity-order system

图2 商品订购系统体系结构

atomiccomponent Tclient() {

port {*portc*1:Tcaccess. *portc*2:Tmaccess.}

computation {

via *portc*1∧*order* **send** orderdata.**via** *portc*1∧*response* **receive** replydate**.**

if *replydata*∧*accept* **then**

```
            {via portc2∧pay send replydata∧payment. via portc2∧confirm receive confirmation}
        replicate}}
atomiccomponent Tmerchant() {
    port { portm1:Tcaccess. portm2:Tmaccess.}
    computation  {
        choose {
            {via portm1∧order receive orderdata.unobservable.via portm1∧response send replydate},
            {via portm2∧pay receive payment.unobservable.via portm2∧confirm send confirmation}}
        replicate  }  }
atomicconnector TOrderLink() {
    port {portc-l1,portl-m1:Tcaccess. portc-l2,portl-m2:Tmaccess.}
    routing  {
        choose  {
            {via portc-l1∧order receive orderdata.via portl-m1∧order send orderdata.
             via portl-m1∧response receive replydata.via portc-l1∧response send replydate },
            {via portc-l2∧pay receive payment.via portl-m1∧pay send payment.
             via portl-m2∧confirm receive confrimation.via portc-l2∧confirm send confirmation}}
        replicate  }  }
compositecomponent  TOrderSystem() {
    instance  {client:Tclient(). merchant:Tmerchant().cmlink:TOrderLink().}
    configuration {
            attach client∧port1 to cmlink∧portc-l1.attach cmlink∧portl-m1 to merchant∧port1.
            attach client∧port2 to cmlink∧portc-l2.attach cmlink∧portl-m2 to merchant∧port2.
            inaction}}
```

上述描述很容易根据表 2 转换为π演算进程,例如,Tmerchant 的行为可表达为进程 $P=(\alpha x.\tau.\overline{\beta}y+\psi z.\tau.\overline{\eta}k).P$.

2.3 动态行为规约及其形式化解释

在 D-ADL 中,动态行为规约是通过 choreographer 来处理的.动态行为本质上是对体系结构的动态重配置,涉及到如下的体系结构变动:(1) 动态创建新的构件实例和连接件实例以及新的端口和通道;(2) 动态删除构件实例和连接件实例以及端口和通道;(3) 体系结构元素之间连接的改变.choreographer 的语法规格如下:

```
choreographer::=attach.choreographer|detach.choreographer|create.choreographer|
                destroy.choreographer|Ecomputation.choreographer|inaction|replicate
detach::=detach ComponenInstance∧Port from ConnectorInstance∧Eport|
        detach ComponenInstance∧Port∧Channel from ConnectoInstance∧Eport∧Echannel
create::=new [01 ComponentInstanc:]ComponentName[01 Actual-parameter list]|
          new [01 ConnectorInstanc:]ConnectorName[01 Actual-parameter list]|
          new [01 Port:]PortTypeName|new [01 Port∧Channel:] ChannelTypeName
destroy::=delete ComponentInstance|delete ConnectorInstance|delete Port|delete Port∧Channel
```

attach 起着建立新连接的作用,语法规格参见第 2.2 节;detach 起着解除连接的作用;create 使用 **new** 关键字动态创建新的构件实例和连接件实例,以及新的端口和通道(其中:Actual-parameter 是实际参数列表;PortTypeName 是端口类型名;ChannelTypeName 是通道类型名);destroy 使用 **delete** 关键字动态删除构件实例和连接件实例以及端口和通道;Ecomputation 是 computation 的扩展版本.由于动态行为以体系结构元素为处理对象,Ecomputation 可以处理包含体系类型数据在内的任意类型数据,而 computation 只能处理常规类型数

据;Eprefix 是 prefix 的扩展版本,它们的区别与 Ecomputation 和 computation 之间的区别是一样的.动态行为规约的高阶π演算解释见表 3.

Table 3 The high-order π-calculus semantic interpretation for the specification of dynamic behavior

表3 动态行为规约的高阶π演算语义解释

D-ADL tag		Interpretation of high-order multi-type π calculus		Remark
Detach	Detach…from…	α operation of re-name		Associated channel is renamed different name
Create	New…*ComponentName*… or New…*ConnectorName*…	P represents the component (connector) behavior, P_a represents the component (connector) computation behavior (route behavior), P_b represents dynamic behavior, P_c represents creating behavior, Q represents the runtime environment behavior, ζ represents the virtual channel, θ represents new port or new channel. $P_{a1},P_{a2},\ldots,P_{an}$ represents the member behavior of composite component, and P_{ak} represents the behavior of new component or member connector.	$P=P_a+P_b$ $P_b=\zeta x.(x\|P_a+P_b)$ $Q=\overline{\zeta}\,P_c.Q$	Add a new component or connector in composite component (connector)
	New…*PortTypeName*… or New…*ChannelTypeName*…		$P=P_a+P_b$ $P_b=\zeta x.(P_a+P_b)$ $Q=\overline{\zeta}\,\theta.Q$	Create a new port or new channel
Destroy	Delete *componentInstance* or Delete *ConnectorInstance*		$P=P_a+P_b$ $P_b=\overline{\zeta}\,P_{ak}.(P_{a1}\|\ldots\|P_{a(k-1)}\|P_{a(k+1)}\|\ldots\|P_{an}+P_b)$ $Q=\zeta x.Q$	Delete a component (connector) from composite component (connector)
	Delete *Port* or Delete *Port*∧*Channel*		$P=P_a+P_b$ $P_b=\overline{\zeta}\,\theta.(P_a+P_b)$ $Q=\zeta x.Q$	Delete a port or channel

对于增删构件实例、连接件实例以及端口和通道的行为,在进行形式语义映射时存在一个难点:π演算没有对应于 **new** 和 **delete** 关键字的操作符,因此不能将它们直接变换为π演算进程.我们的方法是将运行环境的行为纳入来考虑.这是因为在软件实际执行过程中,体系结构的变更行为总是由软件本身和运行环境共同参与完成的,例如构件实例的创建可能需要中间件提供的服务或程序语言运行时库的支持.对运行环境行为的完整描述和形式化是一个复杂的问题,所幸这里只需对运行环境参与上述变更行为的相关动作进行形式化描述即可.

以复合构件增加一个新成员构件实例为例说明(**new**…*ComponentName*…),由表 3 可知下述公式成立:

$$P=P_a+P_b \tag{1}$$

$$P_b=\zeta x.(P_a+P_b) \tag{2}$$

$$Q=\overline{\zeta}\,P_c.Q \tag{3}$$

式(1)说明复合构件行为是其计算行为和动态行为的选择组合;式(2)说明动态行为是首先经由ζ通道输入变量 x,然后执行进程$(x|P_a+P_b)$;式(3)说明运行环境的相关动作序列:反复执行经由ζ通道输出进程 P_c 的动作.由于系统行为是体系结构行为和运行环境相关行为的并行组合,因此系统行为是:

$$P|Q=(P_a+P_b)|\,\overline{\zeta}\,P_c.Q=(P_a+\zeta x.(x|P_a+P_b))|\,\overline{\zeta}\,P_c.Q.$$

按照 R-COM 归约规则,上式可归约为$(P_c|P_a+P_b)|Q$.即复合构件行为演化为$(P_c|P_a+P_b)$,而复合构件的计算行为由 P_a 演化为 $P_c|P_a$,意味着复合构件创建了一个行为是 P_c 的新成员构件实例.仿照以上处理的思路也能将体系结构的其他变更行为解释为高阶π演算进程.

关于如何使用 D-ADL 描述动态行为,我们也以前例进一步说明.假设订购服务器(merchant)发生错误而死机或崩溃时,系统需要自动重新启动一个服务器实例,并将客户请求导向新的服务器,使服务不致中断.这种具有自动切换功能的商品订购系统的体系结构 D-ADL 描述如下:

```
compositecomponent TDynamicOrderSystem() {
    port {environment: Tenvironment.}
        :
        :
    choreographer {
            via environment∧servermessage receive sign.
            if sign=0 then {
```

detach *merchant*∧*port*1 **from** *cmlink*∧*portl-m*1.**detach** *merchant*∧*port*2 **from** cmlink∧*portl-m*2.
delete merchant.
new merchant:Tmerchant().
attach *merchant*∧*port*1 **to** cmlink∧*portl-m*1.**attach** *merchant*∧*port*2 **to** *cmlink*∧*portl-m*2.}
replicate } }

TdynamicOrderSystem 与 TOrderSystem 相比,增加了一个与运行环境进行通信的端口 environment 以及动态行为规约 choreographer.前者是因为检测 merchant 发生错误的任务应由运行环境完成,后者描述订购服务器的动态切换过程:当运行环境检测到 merchant 发生故障时(sign 值为 0),通过 environment∧servermessage 通道向 TDynamicOrderSystem 发送消息.TDynamicOrderSystem 得知消息后,解除 merchant 和 cmlink 之间的连接,然后销毁 merchant,接着创建新的服务器(仍然取名为 merchant),并使之与 cmlink 建立连接.

2.4 D-ADL对系统联机演化和SA求精的形式化支持

使用形式化工具的目标是协助系统设计、验证和演化.π演算理论核心是行为的模拟和等价理论.对于行为模拟和等价概念的详细讨论超出了本文的研究内容,请参见文献[7,8].本节我们使用这些理论来探讨 D-ADL 对系统联机演化和体系结构模型求精的支持.

首先我们使用弱等价概念来讨论系统联机演化时构件替换的正确性.软件系统的各部分相互协作和相互通信,每一个成员都对与它进行协作的成员有一个期望的交互方式和行为约束.这意味着即使两个构件具有不同的内部结构和不同的内部行为,但从任何环境中取出其中一个而放入另外一个,环境不会感知替换发生.换句话说,在软件运行期间,构件之间的相互替换不仅要使得它们的接口保持兼容,而且它们的外部行为也要保持一致.当然,无论是构件还是连接件,其外部可见行为是发生在接口处的交互活动,即在通道处定义的接收(receive)或发送(send)活动,其他活动都被视作内部活动.在π演算中定义了两种行为等价关系:强等价关系和弱等价关系.强等价关系既考虑系统的内部行为,又考虑系统的外部行为;而弱等价关系是一种不考虑系统的内部行为的行为等价理论,意指如果系统 P 和 Q 弱互模拟,则 P 和 Q 表现出来的外部行为是等价的.因此有:

规则 **1**. 设构件 A 和 B 的行为分别被形式化为进程 P_a 和 P_b,则要使得 A 和 B 能够相互联机替换,必要的条件就是 P_a 和 P_b 具备弱等价关系.

D-ADL 以高阶多型π演算作为行为语义基础,因此构件的行为规约能够从 D-ADL 描述出发转换为高阶π演算进程,进而利用规则 1 和高阶π演算弱等价判定理论及其工具推导出构件能否替换.以商品订购系统为例,若客户接收到商家同意其订购请求后,有两种选择:购买商品(向商家发出付款信息)和放弃购买(仅有内部处理动作),则新的客户构件描述如下:

atomiccomponent Tnewclient() {
port {*portc*1:Tcaccess. *portc*2:Tmaccess.}
computation {
via *portc*1∧*order* **send** orderdata.**via** *portc*1∧*response* **receive** replydate.
if *replydata*∧*accept* **then**
{**choose**
{**via** *portc*2∧*pay* **send** *replydata*∧*payment*. **via** *portc*2∧*confirm* **receive** confirmation},
unobservable.}
replicate } }

Tclient 的行为规约和 Tnewclient 的行为规约可分别转换为进程 P 和 Q:

$$P=\overline{\alpha}x.\beta y.[i=1]\overline{\delta}z.\eta h.P\ ,\ Q=\overline{\alpha}x.\beta y.[i=1](\overline{\delta}z.\eta h+\tau).Q\ .$$

根据弱等价判定理论或使用π演算工具 Workbench[10]可推导出 P 弱等价 Q.因此,运行期间,由 Tnewclient

实例化的构件具备替换由 Tclient 实例化的构件的能力.

下面,我们再利用行为弱模拟理论来讨论 SA 行为求精问题.求精是软件开发中的基本活动,在 SA 的构造过程中,如果 SA 的抽象粒度过大,就需要对 SA 进行求精、细化,行为求精又是 SA 求精活动的核心.行为求精的基本原则是维持不同层次行为信息的一致性.具体来说,虽然求精结果可能表现出一些新的行为,但环境对求精对象的行为期望应该被求精结果所维持.使用高阶π演算的行为模拟术语表示,就是求精结果可以弱模拟求精对象的行为,但求精对象无须弱模拟求精结果的行为.这种弱模拟关系在构件行为求精和连接件行为求精上又有不同的具体体现.

在π演算中,弱模拟实际上还可细分为观察弱模拟和分支弱模拟两种,通常所说的弱模拟概念指的是观察弱模拟.观察弱模拟意指对不可观察的活动序列进行抽象,并跟踪每一个可观察的活动.对构件的行为求精,着眼点在于外部观察行为之间的观察弱模拟关系.

规则 **2**. 设进程 P_A 表示构件 A 的行为,进程 P_B 表示构件 B 的行为,若构件 A 是对构件 B 的求精,则 P_A 观察弱模拟 P_B.

分支弱模拟意指对不可观察的活动序列进行抽象,在跟踪每一个可观察的活动的同时保持进程的所有分支轨迹.连接件旨在建立构件间的交互,而交互的核心体现为路由行为,连接件求精应该保持路由行为的一致.

规则 **3**. 设进程 P_C 表示连接件 C 的行为,进程 P_D 表示连接件 D 的行为,若连接件 C 是对连接件 D 的求精,则 P_C 分支弱模拟 P_D.

下面仍以商品订购系统为例说明行为求精规则.在接收到客户订购请求后,商家根据情况确定是否能够满足订购请求的实际过程是订购服务器向仓储服务器查询是否有足够供货.因此,Tmerchant 求精如下:

```
atomiccomponent Tmerchant() {
    port {portm1:Tcaccess. portm2:Tmaccess.portm3:Tinquire}
    computation   {
      choose {
            {via portm1∧order receive orderdata.   via portm3∧inquire send orderdata.
             via portm3∧answer receive result.
             if result then
                {unobservable. via portm1∧response send record(true,payment)}
             else
                {unobservable. via portm1∧response send record(false,0)}},
            {via portm2∧pay receive payment.unobservable.via portm2∧confirm send confirmation}}
      replicate   }   }
```

求精后,Tmerchant 添加了一个新的端口 *portm*3:Tinquire,用于向仓储服务器查询是否有足够供货(inquire).当 Tmerchant 接收到仓储服务器的肯定答复时(result 值为真),将向客户发出同意订购请求和应付款信息;否则拒绝客户请求.求精前后的 Tmerchant 行为规约可分别转换为进程 P 和 Q:

$$P=(\alpha x.\tau.\overline{\beta}y+\psi z.\tau.\overline{\eta}k).P\ ,\ Q=(\alpha x.\overline{\delta}e.\eta f.([f=1]\overline{\beta}g+[f=0]\overline{\beta}h)+\psi z.\tau.\overline{\eta}k).Q\ .$$

根据观察弱模拟判定理论或使用 Workbench 可以推导出 Q 观察弱模拟 P,但 P 并不观察弱模拟 Q.由此可知,求精活动符合规则 2,保持了求精前后行为的一致性.

3 相关工作

动态 ADL 的研究首先体现在将动态配置功能集成在 ADL 中.为开发能随着环境和需求变化而动态自适应的系统,Dowling 等人设计了 K-Component 框架元模型[11].在 K-Component 元模型中,一个有类型的配置图被用来表示 SA,其中节点表示构件接口,类型标签表示构件,边表示连接件.基于关注分离这一软件工程基本原则,尽量使演化逻辑与计算逻辑解耦,K-Component 元模型提供一个专门的机制“adaptation contracts”来显式地表达

演化逻辑,而不将其固化于程序语言或支撑平台之中,从而使得演化逻辑可编程和动态地修改.K-Component 元模型的缺陷是,配置图虽然直观,但并不能严格和全面地表达 SA 的行为语义和结构语义.C2 是一种基于构件和消息的体系结构风格,为体系结构的演化提供了特别的支持[12].C2 的动态管理机制使用专门的体系结构变更语言 AML(architecture modification language).在 AML 中,定义了一组在运行时可插入、删除和重新关联体系结构元素的操作,如 addcomponent,weld 等.但 C2 也没有严密的形式化基础,不能对体系结构的动态行为进行严格的分析和推演.Rapide 是 Luchham 等人开发的体系结构描述语言[13],基于偏序事件集(partially ordered event sets)对构件的计算行为和交互行为进行建模,允许在 where 语句中通过 link 和 unlink 操作符重新建立结构关联.在 Rapide 的连接机制嵌入配置定义中,两者不能分离,Medividovic 将它称为嵌入配置 ADL(in-line configuration ADL)[14].因此,Rapide 不允许单独对连接件进行描述和分析,并且没有提供相关机制,将多个连接机制捆绑成为一个整体,构成复杂的交互模式.因此,Rapide 描述构件交互模式的能力存在不足.

Wright 是 Allen 等人开发的静态 ADL[5],它的形式语义基础是通信顺序进程 CSP(communication sequence process),支持 SA 的静态分析.Dynamic Wright[15]使用标签事件(tagged events)技术对 Wright 进行扩展,从而支持对动态 SA 建模和分析.Dynamic Wright 虽然长于进行诸如死锁检测、模型一致性验证等工作,但对动态行为的表达力不足.同时,基于 CSP 的特点,难以进行行为模拟和等价判定工作.Darwin 是 Magee 和 Kramer 开发的动态 ADL[6],提供延迟实例化(lazy instantiation)和直接动态实例化(direct dynamic instantiation)两种技术,支持动态 SA 建模,允许事先规划好的运行时构件复制、删除和重新绑定.Darwin 虽然运用简单的一阶π演算提供构件计算和交互规约的语义,但其动态机制并不提供任何π演算语义,因此,Darwin 不能提供动态行为形式化分析的基础.

欧盟的 ArchWare[16]项目旨在以体系结构模型为中心构造可演化的软件系统.ArchWare 提出了一种动态体系结构描述语言π-ADL[4].π-ADL 也是一种基于高阶π演算的形式化语言,支持动态体系结构建模和验证.D-ADL 借鉴了π-ADL 的一些思想,但与π-ADL 相比有如下不同:(1) D-ADL 显性定义了体系结构的动态行为操作符“new”和“delete”,π-ADL 则借助π进程的输入输出动作间接实现动态行为.虽然 D-ADL 的动态行为操作符的语义解释也归于π进程的输入输出动作,但从语用学角度看,使用 D-ADL 更便于动态行为的描述和理解;(2) D-ADL 将动态行为与计算行为相分离,使得动态体系结构的行为视图更加清晰,同时由于动态行为具备高阶π演算语义,能够预先推导动态行为的结果,因此可采取措施使这种分离不影响计算行为的正确性;(3) π-ADL 直接支持体系结构的演化(规约变化,通过 Composition/Decomposition 算子实现),D-ADL 本身仅实现了对体系结构动态行为(规约不变)的描述,但对体系结构的演化也有间接的支持(例如第 3.4 节中的有关讨论).

4 总 结

D-ADL 是基于高阶多型π演算的动态体系结构语言.本文介绍了 D-ADL 的描述框架和类型系统,阐述了 D-ADL 中的行为语法规范和语义解释,还探讨了在 D-ADL 规约框架下将π演算的行为模拟和等价理论用于系统联机演化和 SA 模型求精的规则.与其他动态 ADL 相比,D-ADL 具有如下特点和意义:(1) D-ADL 将动态行为从计算行为中分离出来,显式地、集中地表示,便于体系结构动态逻辑的理解、编写和修改;(2) D-ADL 中的动态行为可形式化为高阶π演算进程,因此能够预先推导动态行为的结果;(3) 借助于高阶π演算理论和工具,D-ADL 可用于动态体系结构模型的形式化验证、求精和演化.

进一步的工作是结合其他形式化规约方法,完善 D-ADL 语言.基于高阶π演算理论,D-ADL 适合描述动态系统的结构和行为.然而,体系结构尚有许多方面需要进行描述,如系统的属性和约束.如何运用多种形式化机制,从不同视点描述体系结构,并组成一个有机整体,是 D-ADL 面临的巨大挑战.此外,虽然已经存在许多高阶π演算工具,但并不能够完全满足 D-ADL 的需求.因此,需要进一步研制相关支持工具,对复杂系统规约能够进行自动化分析、检测和仿真.

References:

[1] Oreizy P, Gorlick M, Taylor R. An architecture-based approach to self-adaptive software. IEEE Intelligent Systems, 1999, 14(3):54−62.

[2] Sun CA, Jin MZ, Liu C. Overviews on software architecture research. Journal of Software, 2002,13(7):1229−1237 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/13/1229.pdf

[3] Mei H, Chen F, Feng YD, Yang J. ABC: An architecture based, component oriented approach to software development. Journal of Software, 2003,14(4):721−732 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/721.htm

[4] Oquendo F. π-ADL: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Software Engineering Notes, 2004,29(3):1−14.

[5] Allen R. A formal approach to software architectures [Ph.D. Thesis]. Pittsburgh: Carnegie Mellon University, 1997.

[6] Magee J, Kramer J, Giannakopoulou D. Behaviour analysis of software architectures. In: Donohoe P, ed. Proc. of the 1st Working IFIP Conf. on Software Architecture. Boston: Kluwer Academic Publishers, 1999. 35−50.

[7] Milner R, Parrow J, Walker D. A calculus of mobile processes. Information and Computation, 1992,100(1):1−40.

[8] Sangiorgi D. Expressing mobility in process algebras: First-Order and higher-order paradigms [Ph.D. Thesis]. Edinburgh: University of Edinburgh, 1992.

[9] Li CY. Research on architecture-based software dynamic evolution [Ph.D. Thesis]. Hangzhou: Zhejiang University, 2005 (in Chinese with English abstract).

[10] Victor B, Moller F. The mobility workbench: A tool for the π-calculus. In: Dill D, ed. Proc. of the 6th Int'l Conf. on Computer Aided Verfication. Berlin: Springer-Verlag, 1994. 428−440.

[11] Dowling J, Cahill V, Clarke S. Dynamic software evolution and the k-component model. In: Northrop L, Vlissides J, eds. Workshop on Software Evolution, Conf. on Object-Oriented Programming Systems, Languages, and Applications 2001. New York: ACM Press, 2001.

[12] Oreizy P, Medvidovic N, Taylor RN. Architecture-Based runtime software evolution. In: Nuseibeh B, ed. Proc. of the 2002 Int'l Conf. on Software Engineering. Washington: IEEE Computer Society Press, 1998. 177−186.

[13] Luckham DC, Vera J. An event-based architecture definition language. IEEE Trans. on Software Engineering, 1995,2(9):717−734.

[14] Medvidovic N, Taylor RN. A framework for classifying and comparing architecture description languages. In: Bertolino A, eds. Proc. of the 6th European Software Engineering Conf. LNCS 1301, Springer-Verlag/ACM Press, 1997. 60−76.

[15] Allen R, Douence R, Garlan D. Specifying and analyzing dynamic software architectures. In: Astesiano E, ed. Proc. of the Fundamental Approaches to Software Engineering. LNCS 1382, Berlin: Springer-Verlag, 1998. 21−37.

[16] Oquendo F, Warboys B, eds. ARCHWARE: Architecting evolvable software. In: Oquendo F, ed. 2004 European Workshop on Software Architecture. LNCS 3047, Berlin: Springer-Verlag, 2004. 257−271.

附中文参考文献:

[2] 孙昌爱,金茂忠,刘超.软件体系结构研究综述.软件学报,2002,13(7):1229−1237. http://www.jos.org.cn/1000-9825/13/1229.pdf

[3] 梅宏,陈锋,冯耀东,杨杰.ABC:基于体系结构、面向构件的软件开发方法.软件学报,2003,14(4):721−732. http://www.jos.org.cn/1000-9825/14/721.htm

[9] 李长云.基于体系结构的软件动态演化研究[博士学位论文].杭州:浙江大学,2005.

李长云(1972-),男,湖南耒阳人,博士,副教授,主要研究领域为软件体系结构,软件自动化.

何频捷(1981-),男,硕士生,主要研究领域为软件体系结构,软件框架.

李赣生(1940-),男,教授,博士生导师,主要研究领域为编译理论,软件自动化.