一门新的面向构件语言

陈 波[1], 谭庆平[1], 李舟军[2], 陈火旺[1]
(1. 国防科技大学计算机学院, 湖南长沙 410073; 2. 北京航空航天大学计算机学院, 北京 100083)

摘 要: 作为软件体系结构的重要贡献之一, 连接子显式地描述了构件之间的交互. 本文认为连接子的重要性值得我们在程序设计语言中加以支持. 提出了一门新的面向构件语言 SAJ, 把构件, 端口, 连接子等软件体系结构概念引入到 SAJ 中. 连接子与构件在 SAJ 中都是一阶实体, 使得软件体系结构在底层实现中更加显式化, 而且能更好地支持构件和连接子的复用. 给出了 SAJ 语言的语法、语义和类型系统, 并说明其具有类型安全性.

关键词: 软件体系结构; 连接子; 类型系统
中图分类号: TN311 文献标识码: A 文章编号: 0372-2112 (2006) 12A-2462-05

A New Component-Oriented Programming Language

CHEN Bo[1], TAN Qing-ping[1], LI Zhou-jun[2], CHEN Huo-wang[1]
(1. *School of Computer Science, National University of Defense Technology, Changsha, Hunan 410073, China;*
2. *School of Computer Science, Beihang University, Beijing 100083, China*)

Abstract: The idea of a connector, which explicitly describes the interactions among components, is one of the important contributions of the research on software architecture. In this paper, we argue that such an important abstraction deserves first-class support from programming languages. We proposed a new component-oriented programming language SAJ (Software Architecture-based Java), which integrates some architectural concepts such as the component, the port and particularly the connector into SAJ. The connector is treated as a first-class entity in SAJ as is the component so that software architecture can be made more explicit at implementation level and the simultaneous reuse of the component and the connector can be realized. We formalize our language giving both the type system and operational semantics and show its soundness property.

Key words: software architecture; connector; type system

1 引言

近年来, 软件体系结构的研究得到了广泛的关注和重视. 软件体系结构研究目的在于使软件的体系结构显式化, 它把大型软件系统的高层结构作为研究对象, 将软件系统在较高抽象层次上划分为构件和连接子. 构件是具有相对独立功能的计算单元, 连接子显式地描述了构件之间的关系和交互. 人们逐步认识到对于大型复杂软件系统而言, 系统的功能属性和非功能属性往往取决于构件之间的交互协作, 因此对连接子的研究已经成为软件体系结构研究的重点. 在体系结构描述层次上, 现有体系结构描述语言 (Architectural Description Language, ADL) 都提供了对连接子的支持, 能够对连接子进行形式化描述、分析和验证, 但目前的研究工作对实现阶段如何实现连接子的问题并没有提供一个令人满意的解决方案. 现有的实现连接子的方法包括: 将连接子实现散布在发生交互的各个构件实现之中、采用操作系统的底层机制、采用特殊的构件实现连接子以及上述方法的综合使用. 大多数方法都使得连接子在实现代码中不可见, 不可避免地会导致软件系统的体系结构描述、设计与实现存在不一致性问题, ADL 描述、分析、验证的各种约束和各种非功能属性也不能保证在实现中得到保持.

本文认为, 实现连接子最直接了当的解决方案就是在程序设计语言中提供对连接子的显式支持, 使连接子成为和构件同等地位的一阶实体, 从而使实现阶段的构件、连接子和描述、设计阶段的构件、连接子相对应, 从而提高系统一致性、可跟踪性和可维护性. 如果程序设计语言具有安全的类型系统, 则有利于系统各种性质的推导, 比如通信完整性 (Communication Integrity), 从而使 ADL 描述的各种性质更容易在底层实现中得到保持.

从面向构件的程序设计角度来看, 在程序设计语言层次上显式地支持连接子也是非常重要的, 理由如下: (1) 构件是计算单元, 连接子描述交互, 两者在程序设计语言中有着不同的语义, 起着不同的作用; (2) 显式支持连接子, 能更好更直接地支持通信、控制与计算相分离, 有利于构件、连接子的单独复用; (3) 在复合构件中, 连接子起着组合算子的作用, 比起简单的 Mixin 等组合算子有着更丰富的语义. 而现有的面向构件语言及绝大多数构件模型都不支持连接子的概念, 计算和交互缠绕在构件中, 直接造成了构件的复杂性和功能不明确性. 而且我们认为, 在一定程度上也导致了目前构件定义的多样和混乱. 面向构件程序设计语言要有表征计算的概念-构

收稿日期: 2006-09-22; 修回日期: 2006-12-11
基金项目: 国家自然科学基金 (No. 90104026, No. 60473057, No. 90604007)

件, 也要有表征交互的概念-连接子, 这样才能更直接更有效地支持关注点分离和软件复用.

综上所述, 我们认为连接子的重要性值得我们在程序设计语言层次上进行显式支持, 我们研究、设计、开发了一门面向构件的程序设计语言 SAJ, 将构件和连接子等体系结构概念加入到 SAJ 语言中, 提出了一种新的构件和连接子模型, 给出了 SAJ 语言的语义和类型系统, 并说明其具有类型安全性.

2　SAJ 语言

2.1　设计 SAJ 语言的目的

设计 SAJ 语言的目的在于使得底层实现的构件、连接子能够与软件体系结构描述和设计层面的构件、连接子实现自然映射, 从而保持体系结构描述、设计和实现的一致性. 在分析和研究现有 ADL 基础上, 我们挑选出构件、端口、连接子、角色等体系结构核心概念在 SAJ 语言中加以支持. 同时, 我们的目的还在于提供一个能够更好地支持面向构件软件开发的程序设计语言, 所以在 SAJ 语言的设计中充分考虑了在语言层次上为面向构件(对象)设计原则提供支持, 比如关注点分离原则、接口隔离原则、最少知识原则等. 将在下面章节中详细介绍 SAJ 语言, 首先介绍 SAJ 语言的构件和连接子模型.

2.2　构件和连接子模型

构件(连接子)模型定义了构件(连接子)是什么以及如何构造复合构件(复合连接子). 构件是组装单元, 具有规范的接口和显式的上下文依赖, 构件可以被独立地部署并可以被第三方组装[1]. 构件由端口和服务实现两部分组成, 每个端口表示了构件和外部环境的交互点, 端口包括对外部环境的请求服务接口(Require Interface)和提供给外部环境的提供服务接口(Provide Interface), 显式列出请求服务接口使得构件之间依赖显式化、耦合度降低, 有利于构件功能明确化和理解单个构件. 端口概念则有利于信息隐藏和支持接口隔离原则, 值得注意的是 Java 及其它面向对象语言由于允许向下转型, 其接口概念并不完全支持信息隐藏和接口隔离原则.

连接子描述了构件之间的关系和交互, 连接子由角色和交互两部分组成, 角色表示了参加交互的构件所应有的外部行为, 是对可以扮演角色的构件的抽象: 任何可以扮演某一角色的构件都可以(通过某端口)连接到连接子上参与交互. 交互描述角色如何协作, 起着使交互局部化的作用: 连接到连接子上的构件之间的交互在连接子中进行. 连接子起着使构件之间交互显式化和局部化的作用, 而在传统的程序设计中, 构件之间的交互往往散布在发生交互的各个构件之中, 从而导致构件之间紧耦合而不利于构件复用. 理想情况下, 应该可以实现构件和连接子的单独复用, 即构件可以独立于它所参与的各种交互, 可以在多种连接子中参与交互, 而连接子可以为不同的构件提供相同的通信和协调服务(Unix 中的管道(pipeline)是最具说服力的例子). 连接子表示了交互的模式, 是比构件更具复用潜力的实体. 连接子概念支持了关注点分离原则, 在程序设计语言中显式支持连接子使计算和交互相分离, 使用连接子完成诸如适配、日志、安全等公共服务能使业务逻辑与各种贯穿特性(Crosscuting Feature)服务相分离.

根据连接子与构件之间的关系, 连接子可以分为两种类型: 主动连接子和被动连接子. 两者本质区别在于前者封装了构件之间的控制流和数据流, 即连接子依照交互模式指定的执行顺序调用参与交互的构件的服务, 并在它们之间路由数据, 而不是由构件互相调用服务和传输数据. 主动连接子封装了控制和通信, 构件封装了计算, 通过使用主动连接子将控制与计算分开, 解决了控制和计算混在构件中的问题, 使构件之间耦合度大大降低, 起到了同时复用构件和连接子的作用. 主动连接子表示了多个构件之间长期的关系和交互, 适合于集中式结构的软件系统, 比如工作流系统、企业内部应用. 而被动连接子表示了两个构件之间相对短暂的关系和双向交互. 当构件调用另一构件服务的时候, 被动连接子被创建; 当两个构件交互完成时, 被动连接子被销毁. 被动连接子起着将构件调用服务的请求透明地传递到另一构件相对应的端口并将结果返回的作用, 也就是说连接子作为两个构件之间私有的通信通道起到了传输数据和传递控制的作用. 构件本身封装控制和计算, 使用被动连接子将通信与计算相分离, 在一定程度上也起到了复用构件和连接子的作用. 被动连接子适合应用于分布式结构的软件系统中, 比如 p2p 计算和 web 服务. 我们认为主动连接子与被动连接子是互补的, 分别适用于不同体系结构的应用, 两者区别见表 1.

表 1　主动连接子与被动连接子的区别

	主动连接子	被动连接子
与构件的关系	主动	被动
生命周期	长	大多数情况较短
控制流	封装控制流	控制传递
数据流	知道数据的意义	不知道数据意义
适用环境	集中式结构	分布式结构

SAJ 语言支持复合构件和复合连接子的概念. 复合构件与原子构件的不同在于它内部具有体系结构, 由子构件和连接子组成, 向外部提供的服务由连接子连接起来的各个子构件之间的协作完成. 复合连接子与复合构件一样具有内部体系结构, 不同的是其内部是由实现各种贯穿特性服务的公共服务实现方法集、构件实例、连接子实例组成, 而这些公共服务大多是与业务逻辑无关的, 比如安全、日志等. 我们采用了继承和增量合成的方法实现自定义(复合)连接子. 构件之间的交互无论怎样复杂, 在整体上总是属于某类风格(类型)的[2], 用户自定义连接子通过继承反映这种风格的预定义连接子(SAJ 语言预定义了过程调用(ConProcCall)、黑板(CorrBlackBoard)等连接子), 然后根据需要添加各种公共服务和嵌套连接子、构件来实现复杂的交互和通信协议.

3　SAJ 语言的形式化

在本节中, 我们给出 SAJ 语言的语法、语义和类型系统, 并说明 SAJ 语言具有类型安全性.

3.1　SAJ 语言的语法与语义

SAJ 语言的核心基于 RelJ[3], SAJ 语言在 RelJ 基础上添加了构件、连接子、端口、角色(接口)等软件体系结构概念,

SAJ 语法如图 1 所示, 限于篇幅, 下面仅给出与本文内容相关的若干定义.

定义 **1** 程序 P 由接口、构件类和连接子类组成.

定义 **2** 接口 inf 由二元组(inf', S)定义, inf' 指基接口, S 表示从方法名到方法类型的映射.

定义 **3** 构件类 c 由四元组(c', F, M, Z)定义, c' 指基类, F 表示从字段名f 到字段类型 t 的映射, M 表示从方法名 m 到方法定义的映射, Z 表示从端口名z 到端口类型zt 的映射.

定义 **4** 接子类 d 由五元组$\langle d', r, r', F, M\rangle$定义, d' 指基类, r 和r' 表示参与交互的角色(简单起见, 我们假设每个连接子内有两个角色, 每个连接子连接两个构件), 角色将由二元组$\langle c, z\rangle$扮演, c 表示参与连接的构件类型, z 表示连接到连接子的构件端口, F 和M 同上.

$P\in$ Program	::=intf*com*con*
intf	::=Interface *inf* extends *inf'*{$\overline{m:mt}$}
com	::=Component Class c extends c' {$\overline{t\ f}$, Meth, $\overline{zd}$ }
con	::=Connector Class d extends d'($R\ r, R'\ r'$) {$\overline{t\ f}$,Meth}
Primtype	::=int | boolean|void|....
$n\in$ RefType	::=c|d|*inf* (*interface-name*)
$t\in$ Type	::= Primtype| RefType
Meth	::=$t\ m(\overline{t\ x})\ mb$
$mb\in$MethBody	::={s return e;}
$mt\in$Methtype	::=$\overline{t}\rightarrow t$
R, R', Req, Pro	::=*inf*
zd	::=Port zt {Require Req;Provide Pro;} z
$u, v\in$Value	::=pv|null|l (*address*)
$pv\in$PrimValue	::=true | false |intvalue|....
var	::=x |$e.f$
$ee\in$ Expression	::=v|var|e_1==e_2|se
$se\in$StatementExp	::=var=e|$e.m(\overline{e})$|$e.z.m(\overline{e})$|$e.r.m(\overline{e})$|new c( )|new d (r,r')
$s\in$Statement	::=ε | se;s_1 | if (e) then {s_1} else{s_2};s_3
$rt\in$ Term	::=s|e

图 1 SAJ语言语法

定义 **5** 接口表 I 记录从接口名到接口定义的映射, 构件类表 C 记录从构件类名到构件类定义的映射, 连接子类表 D 记录从连接子类名到连接子类定义的映射. 具体映射如下所示:

$I\in$InterfaceTable	: InterfaceName → InterfaceName × IMethMap
$C\in$ComponentTable	: ComponentClassName → ComponentClassName × FieldMap × MethMap × PortMap
$D\in$ConnectorTable	: ConnectorClassName → ConnectorClassName × RoleType × RoleType × FieldMap × MethMap
$S\in$IMethMap	: IMethName → TypeList × Type
$F\in$FieldMap	: FldName → Type
$M\in$MethMap	: MethName → ParameterList × TypeList × LocalMap × Type × MethBody
$Z\in$PortMap	: PortName → ReqInterface × ProInterface
$L\in$LocalMap	: VarName → Type

我们使用小步操作语义给出 SAJ 语言的语义. 在程序运行时刻, 内存中存在的实例包括构件实例和连接子实例, 构件实例由$\ll c\,|\,f_1: v_1, f_2: v_2, \cdots f_n: v_n\gg$表示, 其中 c 表示构件实例的动态类型, f_i 和 v_i 分别表示字段的名和值. 连接子实例由$\ll d, l_1, z_1, l_2, z_2\,|\,f_1: v_1, f_2: v_2, \cdots f_n: v_n\gg$表示, 其中表示连接子实例的动态类型, l_1 和 l_2 表示参与连接的构件实例地址, z_1 和 z_2 表示构件实例参与连接的端口. 我们使用存储区 σ 存储运行时刻构件实例和连接子实例, 使用存储区 ρ 记录构件实例、连接子实例之间的连接关系(我们称 ρ 为连接关系存储区, 其中的 l 表示连接子实例地址), 使用存储区 λ 存储从局部变量名到值的映射, 如下所示:

$$o\in \text{Instance} \quad ::= \ll c\,|\,f_1: v_1, f_2: v_2, \cdots f_n: v_n\gg \;|\ll d, l_1, z_1, l_2, z_2\,|\,f_1: v_1, f_2: v_2 \cdots f_n: v_n\gg$$

$$\sigma \quad ::= \{\overrightarrow{l\ 0}\,|\,l\in dom(\sigma)\}$$

$$\rho \quad ::= \{l_1\times z_1\times l_2\times \overrightarrow{z_2\ l}\}$$

$$\lambda \quad ::= \{\overrightarrow{x\ v}\}$$

下面给出程序格局及归约关系的定义.

定义 **6** 程序格局 *config* 由五元组$\langle\Gamma, \sigma, \rho, \lambda, s\rangle$组成, 其中 Γ 表示类型环境(从变量到其类型的映射), s 表示代执行语句. 错误格局 *errconfig* 由$\langle\Gamma, \sigma, \rho, \lambda, err\rangle$组成, 表示语句归约中产生了错误 *err*.

定义 **7** 格局归约关系 $\rightsquigarrow$: *config* × (*config* ∪ *errconfig*), 程序的执行由格局间归约$\langle\Gamma, \sigma, \rho, \lambda, s\rangle\rightsquigarrow\langle\Gamma', \sigma', \rho', \lambda', s'\rangle$来描述.

(OSNewCom) $\langle\Gamma,\sigma,\rho,\lambda,\text{New } c(\,)\rangle\rightsquigarrow\langle\Gamma,\sigma[l\mapsto\ll c\,\|\,f_i: Init(FD_c(f_i))\gg],\rho,\lambda,l\rangle$
where $l\notin dom(\sigma_1)$

(OSNewActCon) $\langle\Gamma,\sigma_1,\rho_1,\lambda,\text{New } d(l_1,z_1,l_2,z_2)\rangle\rightsquigarrow\langle\Gamma,\sigma_2,\rho_2,\lambda,l\rangle$
where $\forall i\in\{1\ldots n\}\ \ f_i\in dom(FD_{\rho_1 d})\ \ v_i = Init(FD_{\rho_1 d}(f_i))$
$l\notin dom(\sigma_1)$
$\sigma_2=\sigma_1[l\mapsto\ll d, l_1, z_1, l_2, z_2\,|\,f_1: v_1, \ldots f_n: v_n\gg]$
$\rho_2=\rho_1[(l, z_1, l_2, z_2)\mapsto l]$

(OSCallReqMeth) $\langle\Gamma,\sigma,\rho,\lambda,l_1.z_1.m(\overline{u})\rangle\rightsquigarrow\langle\Gamma,\sigma,\rho,\lambda,l.m(\overline{u})\rangle$
where $\sigma(l_1)=\ll c\,\|\ldots\gg\ \ m\in\{c.z_1.R_1\,|\,z_1\in dom(Z_c)\}$
$\rho(l_1, z_1, \_, \_)=l\Rightarrow l\neq\text{Null}$

(OSCallReqMeth) $\langle\Gamma,\sigma,\rho,\lambda,l_1.z_1.m(\overline{u})\rangle\rightsquigarrow\langle\Gamma,\sigma,\rho,\lambda,l.m(\overline{u})\rangle$
where $\sigma(l_1)=\ll c\,\|\ldots\gg\ \ z_1\in dom(Z_{\rho_1 c})\ \ Z_{\rho_1 c}(z_1)=(Req_1, Pro_1)$
$m\in dom(SD_{\rho_1 Req_1})$
$\rho(l_1, z_1, \_, \_)=l\ \ \ l\neq\text{Null}$

(OSTranReqMeth) $\langle\Gamma,\sigma,\rho,\lambda,l.m(\overline{u})\rangle\rightsquigarrow\langle\Gamma,\sigma,\rho,\lambda,l_2.z_2.m(\overline{u})\rangle$
where $\sigma(l)=\ll d, l_1, z_1, l_2, z_2\,\|\ldots\gg$
$\rho(l_1, z_1, l_2, z_2)=l\ \ l_1\neq\text{Null}\wedge l_2\neq\text{Null}$
$\sigma(l_2)=\ll c'\,\|\ldots\gg\ z_2\in dom(Z_{\rho' c'})\ \ Z_{\rho' c'}(z_2)=(Req_2, Pro_2)$
$m\in dom(SD_{\rho_2 Pro_2})$

(OSCallProMeth) $\langle\Gamma,\sigma,\rho,\lambda,l.z_1.m(\overline{u})\rangle\rightsquigarrow\langle\Gamma,\sigma,\rho,\lambda,l.m(\overline{u})\rangle$
where $\sigma(l)=\ll c\,\|\ldots\gg\ \ z_1\in dom(Z_{\rho c})\ \ Z_{\rho c}(z_1)=(Req_1, Pro_1)$
$m\in dom(SD_{\rho_2 Pro_1})$

(OSCallActConMeth) $\langle\Gamma_1,\sigma,\rho,\lambda_1,l.m(\overline{u})\rangle\rightsquigarrow\langle\Gamma_2,\sigma,\rho,\lambda_2,\{s_1\text{ return}_1;\}\rangle$
where
$\sigma(l)=\ll d, l_1, z_1, l_2, z_2\,\|\ldots\gg\ MD_{\rho_1 d}(m)=(\overline{t\ y}, L, t', s_1\text{ return } e_1)$
$dom(L)=\overline{x}\qquad\overline{u}, x'_{this}, \overline{x'}\in dom(\lambda_1)$
$\Gamma_2=\Gamma_1[\overline{u'\ a\ t}][x'_{this}\ a\ d][\overline{x'\ a\ L(x)}]$
$\lambda_2=\lambda_1[\overline{u'\ a\ u}][x'_{this}\ a\ l][\overline{x'\ a\ Init(L(x))}]$
$s_2=s_1[\overline{u'/y}][x'_{this}/this][\overline{x'/x}]\qquad e_2=e_1[\overline{u'/y}][x'_{this}/this][\overline{x'/x}]$
$s_3=s_2[l_1.z_1/x'_{this}.r][l_2.z_2/x'_{this}.r']\ \ e_3=e_2[l_1.z_1/x'_{this}.r][l_2.z_2/x'_{this}.r']$

图 2 部分规约规则

限于篇幅, 图 2 给出了部分归约规则. 我们用从构件和连接子实例被创建到构件之间通过连接子发生交互的过程来介绍这几个规则. OSNewCom 给出了创建构件实例的归约规则, 新的构件实例被创建时, 将在存储区中分配一新存储单元, 并

给构件实例的字段赋初值(*Init*, *FD* 及其它辅助定义见文献[3]). 创建主动连接子实例的规则 OSNewActCon 与 OSNewCom 类似, 但要在存储区 ρ 中添加新建立的连接关系, 创建被动连接子实例的规则除了 l_1 应由 this 代替以外与 OSNewActCon 相同. 连接子实例被创建后, 它所连接的构件就可以开始交互了. 我们首先考虑由被动连接子连接的构件之间的交互. 构件本身可以调用自己参与连接的端口上的请求服务和提供服务, 前者归约规则由 OSCallReqMeth 给出, 服务请求随即被连接子转发到另一参与连接的构件相对应的端口(规则 OSTrarrReqMeth). 另一构件在端口得到服务的请求后找到实现该服务的方法(规则 OSCallProMeth), 根据方法名找到实现体后执行实现体(这与构件调用自己的提供服务的过程一样), 主动连接子的方法调用规则 OSCallActConMeth 与之类似, 在此仅给出了 OSCallActConMeth 规则. 主动连接子方法的实现体中, 对通过端口扮演角色的构件只能调用其连接端口上提供的服务, 不能调用不在连接端口上(可能在其他端口上)提供的服务, 也不能调用端口上请求的服务.

3.2 SAJ 语言的类型系统

在 SAJ 语言中, Component 是构件类的根类, Connector 是连接子类的根类, 主动连接子类 ActCon 和被动连接子类 PasCon 是 Connector 的子类. 子类型推导规则因为比较简单, 在此省略. 与本文有关的几个推导规则如图 3 所示.

(TSNewActCon)	(TSNewPasCon)	(TSCallPortMeth)
$d \in dom(D_p)$	$d \in dom(D_p)$	$\Gamma \vdash e:c \quad \Gamma \vdash \vec{e}:\vec{t}$
$D_p(d)=(d',r,r',F,M)$	$D_p(d)=(d',r,r',F,M)$	$c \in dom(C_p) \quad z \in dom(Z_c)$
$\Gamma \vdash e_1:c_1 \quad \Gamma \vdash e_2:c_2$	$\Gamma \vdash e_1:c_1 \quad \Gamma \vdash e_2:c_2$	$m \in c.z.R \Rightarrow e=this$
$\Gamma \vdash c_1.z_1.R_1 \le r \quad \Gamma \vdash c_2.z_2.R_2 \le r'$	$\Gamma \vdash c_1.z_1.R_1 \le c_2.z_2.R_2$	$mtype(m,c,z)=\vec{i} \to t$
$\Gamma \vdash r' \le c_1.z_1.R_1 \quad \Gamma \vdash r \le c_2.z_2.R_2$	$\Gamma \vdash c_2.z_2.R_2 \le c_1.z_1.R_1$	$\vec{t} \le \vec{i}$
$\Gamma \vdash New\ d(e_1.z_1,e_2,z_2):d$	$\Gamma \vdash New\ d(e_1.z_1,e_2,z_2):d$	$\Gamma \vdash e.z.m(\vec{e}):t$

图 3　几个类型推导规则

TSNewActCon 给出了创建主动连接子实例表达式的类型推导规则. 连接子类名 d 属于连接子类表说明是合法的连接子类. 在新连接子实例创建时, 如果该连接子实例连接的两个构件实例在其连接端口上提供的服务分别是连接子中角色类型的子类型, 说明这两个构件实例可以扮演连接子中的角色, 而且各自在其连接端口上的请求服务都可以被对方连接端口上提供的服务所满足, 则该新连接子实例创建成功, 具有类型 d. 被动连接子起着传递服务请求和传输数据的作用, 所以创建被动连接子实例的表达式的类型推导规则 TSNewPasCon 仅仅要求参与连接的两个构件实例在其连接端口上的请求服务能被对方连接端口上提供的服务所满足. TSCallPortMeth 给出了调用构件端口服务的表达式的推导规则, 如果是调用请求服务, TSCallPortMeth 保证调用请求服务的调用者必须是构件本身.

进一步, 我们给出构件类、连接子类、程序良构的规则. 为此, 我们必须首先给出构件(连接子)类中字段、方法和构件类中端口良构的规则(限于篇幅, 仅仅给出与本文内容有关的端口良构规则, 所有良构规则的形式化描述详见文献[4])、构件类良构规则、主动连接子类良构规则、被动连接子类良构规则和程序良构规则.

规则 **1**　端口良构规则: 在程序 P 中, 对于构件类 c, 如果 c 中端口 z 的请求服务接口和提供服务接口交集为空, 构件类 c 实现了端口 z 上的提供服务, 且如果 c 的基类不是 Component, 那么端口 z 属于基类(换句话说, 只有 Component 的子类可以定义新端口), 则端口 z 是良构的.

规则 **2**　构件类良构规则: 在程序 P 中, 对于构件类 c, 如果其基类是良构的, 且对于构件类中所有的字段 f, 方法 m, 端口 z 是良构的, 则 c 是良构的.

规则 **3**　主动连接子类良构规则: 在程序 P 中, 对于主动连接子类 d, 如果其基类是良构的, 如果 d 的基类不属于 SAJ 预定义的连接子类, 则 d 不能定义新角色, 且对于连接子类中所有的字段 f, 方法 m 是良构的, 则 d 是良构的.

规则 **4**　被动连接子类良构规则: 在程序 P 中, 对于主动连接子类 d, 如果其基类是良构的, 且对于连接子类中所有的字段 f, 方法 m 是良构的, 则 d 是良构的.

规则 **5**　程序良构规则: 在程序中, 所有构件类和连接子类都是良构的, 则程序是良构的.

至此, 我们给出了代码级的类型推导规则. 最后, 我们给出程序运行时刻, 存储区 σ、连接关系存储区 ρ、局部变量映射 λ 及程序格局 *config* 良构的规则. 为此, 我们必须先给出构件实例字段、构件实例和连接子实例良构的规则.

规则 **6**　构件(连接子)实例字段良构规则: 在程序 P 运行时, 如果构件(连接子)实例 o 某字段 f 的值类型是该字段在构件(连接子)类声明的类型, 则该字段在构件(连接子)实例中是良构的.

规则 **7**　构件实例良构规则: 在程序 P 运行时, 如果构件实例中所有的字段是良构的, 则构件实例是良构的.

规则 **8**　主动连接子实例良构规则: 在程序 P 中, 如果连接子实例中所有字段是良构的, 连接子连接的两个构件实例是良构的且两个构件实例通过各自连接端口能分别扮演连接子中的角色, 则连接子实例是良构的.

规则 **9**　被动连接子实例良构规则: 在程序 P 中, 如果连接子实例中所有字段是良构的, 则连接子实例是良构的.

规则 10、11、12 给出了运行时刻各个存储区的良构规则, 规则 13 给出了程序格局良构的规则.

规则 **10**　存储区 σ 良构规则: 在程序 P 中, 如果存储区 σ 中的所有实例(构件实例和连接子实例)是良构的, 则存储区 σ 是良构的.

规则 **11**　连接关系存储区 ρ 良构规则: 在程序 P 中, 如果在连接关系存储区 ρ 中, 对于所有的连接关系, 如果构件实例 $\sigma(l_1)$ 和 $\sigma(l_2)$ 是良构的且连接子实例 $\sigma(\rho(l_1, z_1, l_2, z_2))$ 是良构的, 则存储区 ρ 是良构的.

规则 **12**　局部变量存储区 λ 良构规则: 在程序 P 中, 对于类型环境 Γ 中所有的变量 x, 如果 $\lambda(x)$ 值的类型是 $\Gamma(x)$, 则 λ 是良构的.

规则 **13**　格局 *config* 良构规则: 对于格局 $\langle \Gamma, \sigma, \rho, \lambda, s \rangle$, 如果 σ, ρ, λ 是良构的, s 是良类型的, 则格局 $\langle \Gamma, \sigma, \rho, \lambda, s \rangle$ 是良构的.

3.3 SAJ 语言类型系统的安全性

定理(保持)　在良构程序 P 中, 如果当前格局 $\langle \Gamma, \sigma, \rho,$

λ, s〉是良构的, 那么执行 s 归约到的新格局〈Γ′, σ′, ρ′, λ′, s′〉也是良构的, 而且所有在 σ 中的实例类型在 σ′ 中保持不变.

证明: 限于篇幅, 证略.

定理(进展). 在良构程序 P 中, 所有的良构格局〈Γ, σ, ρ, λ, s〉或者归约到一个新的良构格局〈Γ′, σ′, ρ′, λ′, s′〉, 或者归约到一个错误格局.

证明: 限于篇幅, 证略.

由以上定理可以得出, SAJ 语言的类型系统具有安全性, 从而保证良构的程序在执行过程中不会到达受阻格局.

4 相关工作

在软件体系结构和面向构件的程序设计研究领域, 连接子的研究开始受到普遍的关注和重视, 但大多数的研究集中在高层描述和性质验证, 对于如何在底层实现连接子这方面的研究较少. 文献[5]提出了在程序设计语言上提供对自定义连接子的支持, 在构件语言 ArchJava 基础上使用反射机制来实现连接子抽象. ArchJava 反射库提供了连接子基类, 用户自定义连接子从基类派生, 并重写表征连接子动态语义的 Invoke 函数. 文献[5]进一步指出通过使用 ArchJava 的连接子抽象可以实现文献[2]中列举的各种类型的连接子, 说明了在程序设计语言层次提供对连接子的支持灵活性好, 能很好地提高连接子的复用. 文献[5]存在的问题在于使用了反射机制来实现连接子抽象, 很难保证类型安全性. 而且文献[5]主要目的在于使体系结构描述、设计与底层实现相一致, 没有考虑连接子引入到程序设计语言中所起的作用.

文献[6]提出了 exogenous connector 的概念, 旨在使用将控制和计算分离(exogenous connector 封装控制, 构件封装计算), 使构件之间耦合度降低, 从而起到复用构件和连接子的作用. 我们的主动连接子与 exogenous connector 类似, 但我们认为单单主动连接子是不够的, 它适用的应用有限, 特别是不适合应用在分布式环境下对数据安全敏感的应用, 为此我们提出了被动连接子的概念与之互补.

文献[7, 8]也提出了显式支持连接子的思路. 我们认为, 与软件系统联系最紧密、最直接的是程序设计语言, 在程序设计语言层次上支持连接子比其他方法更灵活、更直接, 所以我们采取了直接在程序设计语言层次上支持连接子和构件的方法. 而且我们还给出了 SAJ 语言的操作语义和类型系统, 比以上工作更加形式化, 为进一步的研究奠定了基础.

5 结束语

本文论述了无论从软件体系结构角度还是从面向构件程序设计语言角度, 连接子都值得我们在程序设计语言层次上进行显式支持. 我们研究、设计和开发了 SAJ 语言, 给出了 SAJ 语言的语法、语义和类型系统. 在 SAJ 语言中, 构件和连接子都是一阶实体, 使得底层实现的构件、连接子可以和软件体系结构描述、设计的构件、连接子相对应, 从而保持体系结构描述、设计和实现的一致性. 而且在程序设计语言层次上显式支持连接子也有利于构件和连接子的复用, 可以更好地支持面向构件的程序设计. 本文还提出了主动连接子和被动连接子的概念, 分析了两者的特点及应用环境.

参考文献:

[1] C Szyperski. Component Software: Beyond Object-Oriented Programming[M]. Addisorr-Wesley, 2003.

[2] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors[A]. In Proc. International Conference on Software Engineering[C]. New York: ACM Press, 2000. 178-187.

[3] Gavin Bierman, Alisdair Wren. First-class relationships in an object-oriented language[A]. In Proc. European Conference on Object-Oriented Programming[C]. Glasgow: Springer, 2005, 262-286.

[4] 陈波. 基于软件体系结构的构件模型和语言研究[D]. 湖南长沙: 国防科技大学, 2006.
Chen Bo. Research on Software Architecture-Based Component Model and Language[D]. Changsha, Hunan: National University of Defense Technology, 2006. (in Chinese)

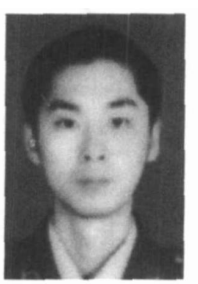
[5] Aldrich J, Sazawal V, Chambers C, Notkin D. Language support for connector abstractions[A]. In Proc. European Conference on Object-Oriented Programming[C]. Darmstadt: Springer, 2003. 74-102.

[6] Kung-Kiu Lau, Perla Velasco Elizondo, ZhengWang. Exogenous connectors for software components[A]. In Proc. Eighth International SIGSOFT Symposium on Component-based Software Engineering[C]. St. Louis: Springer, 2005. 90-106.

[7] Cheoljoo Jeong and Sangduk Lee. Implementing software connectors through first-class methods[A]. In Proc. IEEE conference on system, man, and cybernetics[C]. Denver: IEEE, 2000. 92-96.

[8] Oussalah M, Smeda A, Khammaci T. An explicit definition of connectors for component-based software architecture[A]. In Proc. IEEE International Conference on the Engineering of Computer Based Systems[C]. Brno: IEEE, 2004. 44-49.

作者简介:



陈 波 男, 1976 年出生于山东省济南市, 现为国防科学技术大学计算机学院博士生, 主要研究方向为软件体系结构、程序设计语言. E-mail: chenbo@nudt.edu.cn