ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.17, No.6, June 2006, pp.1411−1422
DOI: 10.1360/jos171411

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

模型驱动架构中模型构造与集成策略*

刘 静[1+], 何积丰[1], 缪淮扣[2]

[1](华东师范大学 软件学院,上海 200062)

[2](上海大学 计算机科学与工程学院,上海 200072)

A Strategy for Model Construction and Integration in MDA

LIU Jing[1+], HE Ji-Feng[1], MIAO Huai-Kou[2]

[1](Software Engineering Institute, East China Normal University, Shanghai 200062, China)

[2](School of Computer Science and Technology, Shanghai University, Shanghai 200072, China)

+ Corresponding author: Phn: +86-21-62232554, E-mail: jifeng@sei.ecnu.edu.cn, http://www.sei.ecnu.edu.cn

Liu J, He JF, Miao HK. A strategy for model construction and integration in MDA. *Journal of Software*, 2006,17(6):1411−1422. http://www.jos.org.cn/1000-9825/17/1411.htm

Abstract: At the core of MDD (model driven development) are the concepts of model and its transformation and refinement. Unified modeling language (UML) is selected by object management group (OMG) as a standard modeling language and model driven architecture (MDA) is constructed on it. However, UML models are not precisely described, especially in semantics. Thus the models developed in different phases or constructed in different views are not easily to be integrated together in MDA. In this paper, based on Hoare and He's unifying theories of programming (UTP), a method is proposed to combine the refinement calculus of object systems (rCOS) with UML is proposed to increase the precision and transformation ability of the models. Models at different abstract levels and different views are constructed and integrated together to form a unified modeling system.
Key words: model driven architecture (MDA); UTP; model; component; formal method

摘 要: 模型驱动式开发是以模型构造、模型转换和精化为核心的.对象管理组织 OMG 选择将统一建模语言 UML 作为标准建模语言,并将其作为模型驱动架构 MDA 的支持平台.但 UML 模型缺乏严格的语义,不同视角和不同开发阶段的模型很难集成,所以目前 MDA 中各种模型之间是脱节的.基于统一程序设计理论 UTP,将对象精化演算系统 rCOS 与 UML 结合起来,提高了 UML 模型的精确性与模型转换能力.建立模型驱动式开发中不同层次、不同视角的相关模型并将它们集成起来,建立了一个统一的模型系统.
关键词: 模型驱动架构(MDA);UTP;模型;构件;形式方法
中图法分类号: TP301 文献标识码: A

模型驱动式软件开发(model driven development,简称 MDD)就是对实际问题进行建模,并转换、精化模型,

* Supported by the National Natural Science Foundation of China under Grant No.60373032 (国家自然科学基金); the National Grand Fundamental Research 973 Program of China under Grant Nos.2002CB312001, 2005CB321904 (国家重点基础研究发展规划(973)); the Natural Science Foundation of Shanghai of China under Grant No.05ZR14052 (上海市自然科学基金)

Received 2006-01-18; Accepted 2006-03-28

直至生成可执行代码的过程[1].模型驱动架构(model driven architecture,简称 MDA)是一种对业务逻辑建立抽象模型,然后由抽象模型自动产生最终完备的应用程序的方法.MDA 致力于提高软件开发行为的抽象级别,倡导将业务逻辑定义为精确的高层抽象模型,让开发人员从繁琐、重复的低级劳动中解脱出来,更多地关注业务逻辑层面[2].它代表了对象管理组织(object management group,简称 OMG)定义的互操作性规范的一个革命性进步.

在 MDA 中,模型不再仅仅是描绘系统、辅助沟通的工具,而是软件开发的核心和主干.模型之间通过模型映射机制相互转换,从而保证了模型的可追溯性.软件的开发和更新过程就是模型自顶而下、逐步精化的过程.MDA 的基本思想是,一切都是模型.软件的生命周期就是以模型为载体并由模型转换来驱动的过程.模型构造、模型映射与模型精化技术是 MDA 的核心[3].

MDA 要求模型是对系统的一部分结构、功能或行为的形式规范:首先,模型是一种系统规范,这种规范可以是结构的规范,也可以是系统功能或系统行为的规范;其次,这种规范必须是形式化的,即必须使用一种严格定义没有歧义的语言.所以,一个模型必须和一种严格定义了语法和语义的建模语言绑定在一起.MDA 展现了一个全面的体系结构,但目前它在各个模块之间是脱节的[4],或者说没有可靠的连接手段.如何建立从需求分析到设计直至实现的各种层次的软件模型,如何通过模型的精化从上一层模型获得下一层模型,仍然是软件工程面临的巨大挑战.

OMG 将 UML 作为标准建模语言为用户提供可视化的建模环境和工具,并将 UML 作为 MDA 支持平台,利用 UML 来建模[5].UML 提供了大量可扩展的预定义结构、半形式化的定义和支持工具.但是,UML 模型缺乏严格的语义[6],因此带来了正确性和一致性方面的问题.设计和建立精确而高效的模型定义语言是当前的重点.UML 2.0 是专门针对 MDA 需求定制的[5,7].但 UML 2.0 的语义还是比较松散的,它用较为形式化的语言 OCL(object constraint language)和自然语言两种手段描述静态语义,而动态语义却是基本上完全用自然语言来描述的.UML 的编制者在规范中承认,尽管用一种精确的方式描述动态语义可以更容易地理解,但实际上 UML 系统却仍然使用自然语言的方式描述动态语义[7].在 MDA 中,非形式化的规范不能作为一个模型.MDA 模型必须是非模糊的.所以需要一种在语法和语义上都精确的建模语言,以跟踪所有模型和模型的更改,并保证软件生命周期中不同时期的模型一致.

在用 UML 建模、模型转换以及提高模型的精确性和一致性方面,已有一些专家、学者做了大量的工作,Garlan 和 Kompanek 分析并评价了用 UML 建模软件体系结构的方式与性能[8];Giese 等人分析了体系结构中相关方面(non-orthogonal aspects)动态行为的形式描述与组合方法[9];Paige 等人将程序设计演算与部分 UML 模型结合起来[10];Amalio 等人将部分 UML 模型形式化[11];Kim 和 Carrington 等人将部分 UML 视图映射到形式语言 Z,并用以描述面向对象的设计模式[12,13].但这些工作大都是将部分 UML 视图形式化,或者是针对软件设计某些阶段、某些方面的需要,不能对整个软件开发过程提供支持.我们的策略是将基于构件的软件开发中的基本模型集成起来,以支持软件开发全过程.

我们在此领域进行过一些相关的研究:研究如何用对象精化演算系统(refinement calculus of object systems,简称 rCOS)来增强 UML 的模型转换能力[14−16];研究将不同视角、不同抽象层次的 UML 模型统一起来的方法[17];研究用 UML 建立关系型需求模型的策略[18];研究如何通过模型转换从需求模型获得设计模型[19],并保证动态需求模型与静态需求模型之间一致的方法[20];研究基于构件的可重用的软件开发架构[21].

本文基于 Tony Hoare 和何积丰教授的统一程序设计理论(unifying theories of programming,简称 UTP)[22]将 rCOS 与 UML 结合起来,以提高模型的精确性,方便模型转换.我们建立了类模型和构件模型等,并给出了模型的语义与模型转换、精化策略;将基于构件的软件开发中涉及的不同模型集成起来,以解决 MDD 中模型之间脱节的问题.

1 模型的构造

在模型驱动式开发中,开发人员首先建立系统模型,按照一定的体系结构风格将构件与连接器组织起来.构

件与连接器模型是通过静态类模型与动态交互模型实现的[7].我们先定义类模型与交互模型,再定义构件模型与软件体系结构.

1.1 类模型

从抽象意义上看,类模型由一个类图和一系列状态断言组成.类图显示了一系列类、属性、方法和关联.我们将类图定义如下:

定义 **1.1**. 类图是多元组

$$\Re=\langle CN,Att,Ass,\lhd\!\!-,Meth\rangle.$$

其中:CN 是类名集合;Att 为 CN 中的每个类 C 分配属性集合$\{T_1a_1,\ldots,T_na_n\}$;Ass 表示关联集合,是 $CN\times AN\times CN$ 的子集,每个五元组$\langle M_C,C,A,D,M_D\rangle$表示类 C 和类 D 之间的关联 A,M_C 和 M_D 是自然数集合,分别表示 A 的角色 C 和 D 的多重性,AN 为关联名称的集合,关联 A 是部分函数,$A:C\nrightarrow(AN\nrightarrow PN\times C\times PN\times\{direct,undirect\})$,其中 PN 是自然数 N 的幂集,$A\in AN$,如果 $A(C)=\langle M_C,D,M_D,direct\rangle$,则 A 是从 C 到 D 的有方向的关联,如果 $A(C)=\langle M_C,D,M_D,undirect\rangle$,$A$ 是 C 与 D 之间无方向的关联.$Meth$ 是从 CN 到类图中所有类的方法的集合的映射,格式为 $m(paras)\{c\}$,$paras$ 是一系列指定类型的参数;$\lhd\!\!-$表示类之间的继承关系,$\lhd\!\!-\subseteq CN\times CN$.

我们将多重关联的规范放在类模型的状态断言中[18].在文献[23]中,我们用 Mana 的转换系统将 UML 概念类图转换成类似于 Java 的形式演算系统 rCOS[14]中的类声明部分.本文在概念类模型中加入方法,将概念类模型与设计类模型统一表示为

Class N **extends** M $\{T\ a$; // for each attribute $T\ a$ in $Att(N)$

Method $m_1(paras_1)\{c_1\}$; … ; $m_i(paras_i)\{c_i\};\}$ // for each m_k in $Meth(N)$

其中:M 和 N 为 CN 中类,M 是 N 的父类;$Att(N)$是类 N 中所有属性的集合;$Meth(N)$是类 N 中所有方法的集合;a 是类 N 的属性,T 是 a 的类型,$a\in Att(N)$;$m_k(paras_k)$是类 N 中的方法,$m_k(paras_k)\in Meth(N)$,$paras_k$ 是该方法的参数.

由于在最初的需求模型(UML 概念模型)中关联是无方向的,但在 UML 设计模型中是有方向的,所以我们定义两种类型的关联——无方向的关联和有方向的关联,供不同的开发阶段使用.

- 对于每一个无方向的关联,C 和 D 是关联 A 所关联的两个类,我们声明:

Class $A\{$ $PC\ c$; // the role C of A

$PD\ d$; // the role D of $A\}$

其中,c和d是类A的属性,分别表示类图中类C和类D的角色.类A还可以表示为$C\times D$.

- 对于有方向的关联,若关联的方向是 C 到 D,我们将 A 声明为 C 的属性.如果关联 A 中 D 是多重的,A 的类型为 D 的幂集,否则为 D:

Class C **extends** M $\{$ //*M is the direct super Class of C;

$T\ a$; //* for each $T\ a$ in $Att(C)$

$PD\ A;\}$ //* or $D\ A$ depending on multiplicity

对象图表示的是变量的状态

$$\sigma=\{O_C,O_A|C \text{ and } A \text{ are class and association names}\}.$$

这里,O_C 记录当前状态下类 C 的所有对象的集合;而 O_A 记录关联 A 对应的所有连接的集合.

对于关联 $A:P(O_{C_1}\times O_{C_2})$和类 C_i,i=1,2 的对象 O_i,我们用 $A(O_i)$表示与 O_{C_i} 相关的类中的对象集合:

$$A(o_i)=_{df}\{o_{i\oplus1}|o_{i\oplus1}\ O_{C_{i\oplus1}}\wedge(o_1,o_2)\in A\},$$

$$A(O_{C_i})=_{df}\cup_{o\in O_{C_i}}A(o).$$

其中 1⊕1=2,2⊕1=1.现在,关联中角色的多重性可以定义为一种状态特性.设 M_1 和 M_2 是 Int 的子集,我们将 M_1,M_2 作为 O_{C_1} (或 O_{C_2})到 O_{C_2} (或 O_{C_1})的连接.

$$Multiplicity(A)=_{df}\wedge_{i=1,2}\forall oi\in O_{C_i}\ (|A(oi)|M_{i\oplus1}).$$

它还要求关联 A 只连接当前状态下存在的对象:对于 D 中的每个关联 $A:P(O_{C_1}\times O_{C_2})$,

$$LinkObjects(A)=_{df}A(O_{C_1})\subseteq O_{C_1}\wedge A(O_{C_2})\subseteq O_{C_2}.$$

如果一个类图中的每个关联 A 都满足条件 $Multiplicity(A)$和 $LinkObjects(A)$,那么该类图的状态是有效的.在后面的讨论中,状态指的是有效的状态.条件 $Multiplicity(A)$和 $LinkObjects(A)$精确地定义了关联 A 的含义和 UML 类图中描述的多重角色.但仅有类、关联及其多重性是不能表达应用程序所需的所有约束的.特别是多重性不允许表达关联之间的关联,我们用关系逻辑来表达状态约束.

一般说来,类模型 $CM=\langle\Delta,\Phi\rangle$,其中$\Delta$是类图,$\Phi$是类和关联上的状态约束.类模型的状态特性$\xi$可以由$\theta\wedge\Phi\rightarrow\xi$推出,即$\xi$在关联计算中可以由$\theta$和$\Phi$导出.我们用 $CM\vdash\xi$来表示 CM 满足ξ,还可以用此方法来定义满足某种状态约束的概念类图之间的转换.

1.2 交互模型

UML 中交互模型包括顺序图与协作图,两者在语义上是等价的,可以互相转换.顺序图由对象和消息组成,描述对象之间是如何通信的.对象之间的交互出现在一个对象调用另一个对象的方法时.

定义 **1.2**. 假设Ω是一个偏序集合.消息是多元组

$$Msg=\langle\underline{Ns},action,\underline{Mt},sord,rord\rangle.$$

其中:$\underline{Ns}$ 表示类 N 的对象 $\underline{s}$,作为消息源;$action$ 是条件命令调用,形式为 $g\rightarrow act$,其中 g 为执行 act 的条件;$\underline{Mt}$ 表示消息的目标对象 t;$sord$ 和 $rord$ 是集合Ω中的元素,分别表示顺序图中消息调用顺序和返回顺序,要求满足 $sord<rord$.

对象 s 可以表示为 $source(Msg)$;$action$ 表示为 $Action(Msg)$;g 是 $source(Msg)$的属性的布尔表达式;act 可能是一条命令,可能是方法调用 $target(Msg).m(paras)$.$\underline{Mt}$ 表示消息的目标对象 t,t 的类型为 M,可表示成 $target(Msg)$;$sord$ 和 $rord$ 分别表示为 $Sord(Msg)$和 $Rord(Msg)$,我们要求 $sord<rord$.例如:消息 Msg_i 的调用序列号为 $Sord(Msg_i)$,返回序列号为 $Rord(Msg_i)$,则 $Sord(Msg_i)<Rord(Msg_i)$.

当条件 g 为假(false)时,条件命令 $g\rightarrow act$ 中的操作 act 不被执行.我们要求:如果 $Action(Msg)$是一个不带方法调用的条件命令,则 $source(Msg)=target(Msg)$,如对象间的交互仅通过方法调用来实现.

定义 **1.3**. 顺序图是多元组

$$Sd=\langle\underline{Ss},Start,MSG\rangle.$$

其中:$\underline{Ss}$ 是调用消息 $Start$ 的初始对象,$Start$ 是开始消息,$target(Start)=\underline{Ss}$.如果 Msg_1 和 Msg_2 来自同一个源对象,$Rord(Msg_1)<Sord(Msg_2)$,且图中没有消息 Msg 使得 $Rord(Msg_1)<Sord(Msg)<Sord(Msg_2)$,则 Msg_2 直接跟在 Msg_1 后面;如果 $target(Msg_1)=source(Msg_2)$,$Sord(Msg_1)<Sord(Msg_2)$且 $Rord(Msg_1)>Rord(Msg_2)$,并且图中不存在消息 Msg 使得 $Sord(Msg_1)<Sord(Msg)<Sord(Msg_2)$,则称 Msg_1 直接调用 Msg_2.MSG 是消息集合,对于该消息集合 MSG 中的每条消息 Msg 都有 $Sord(Start)<Sord(Msg)$,并且有且仅有一条消息 Msg 使得 Msg 直接调用 Msg.

$\underline{Ss}$一般是执行者(actor).通常,开始消息中的目标对象是一个用例句柄,操作是句柄中的方法.为了简单起见,我们不处理循环.循环可以通过定义对顺序图的循环引用来完成.我们用 rCOS 来形式化顺序图:对于每个对象 $\underline{o:N}$ 和消息 Msg,使得 $target(Msg)=\underline{o:N}$,$Action(Msg)$是条件方法 $g\rightarrow a.m(paras)$,其中 g 是执行 a 中的方法 m 的条件,$paras$ 是方法 m 的参数.我们用 $MS(Msg)$表示消息序列 $Sm=Msg_1,\ldots,Msg_k$ 使得 Msg_1 是直接被 Msg 调用的消息;Msg_{i+1} 直接跟在 Msg_i 的后面;顺序图中没有消息直接跟在 Msg_k 后面.

另外,同一条消息可能触发不同的消息序列.

令 $Act(Sm)=Action(Msg_1);\ldots;Action(Msg_k)$,然后在类 N 中定义方法 $m(paras)$:

Class $N::m(paras)\{\sqcap_{Sm\in MS(Msg)}Act(Sm)\}$

顺序图 Sd 的语义定义是出现在 Sd 中的所有方法的总和.

1.3 构件模型

由于 OMG 组织将构件和连接器作为系统构造单元 UML 2.0 中体系结构框架中的标准元素[24],我们将构件和连接器作为系统的组成单元,每个构件包含特定的接口和实现内核.

定义 **1.4**. 构件是系统的组成单元,它由构件接口和构件实现模块组成:

$$Component=\langle Intf,M,A,MImpl,Init\rangle.$$

其中:*Intf* 是构件所有接口的集合;*M* 是类模型;*A* 是 *M* 中已声明类型的所有字段集合;*MImpl* 将接口 *Intf* 中声明的操作映射到(α,*Q*),*Q* 是用 rCOS 语言书写的命令,α是 *A* 中包含的变量,是接口 *Intf* 中描述的操作的输入/输出参数;*Init* 是构件的初始状态.

构件接口集合 *Intf* 可写成 $Intf=\langle intf_1,intf_2,\ldots,intf_n\rangle$,每一个接口 $intf_i$ 是独立的,其属性和行为与其他接口无关,我们将其定义为一个八元组

$$intf_i=\langle ID,Prov_i,Requ_i,Priv_i,Beha_i,Msgs_i,Cons_i,Nonf_i\rangle.$$

其中:*ID* 是构件的标识;$Prov_i$ 是构件的第 *i* 个接口提供给环境或其他构件的功能的集合;$Requ_i$ 是构件的第 *i* 个接口运行所需环境或其他构件功能的集合;$Priv_i$ 是构件的第 *i* 个接口私有属性的集合;$Beha_i$ 是构件的第 *i* 个接口行为语义描述;$Msgs_i$ 是构件的第 *i* 个接口所产生消息的集合;$Cons_i$ 是构件的第 *i* 个接口行为约束,通常包括构件运行的初始条件、前置条件和后置条件.$Nonf_i$ 是构件的第 *i* 个接口非功能说明,包括构件的安全性、可靠性等说明.

构件的实现部分由构件的设计类模型与交互模型描述,接口给出了构件的属性与行为描述.

1.4 连接器模型

连接器是软件体系结构的一个组成部分,它通过对构件间的交互规则的建模来实现构件间的连接.与构件不同,连接器不需要编译,常见的连接器有客户/服务器协议(client/server protocols)、管道(pipe)、过程调用(procedure call)等.作为一种重要软件体系结构组成要素,构件连接器实现了构件间的连接与交互[25,26].本文从行为、角色、消息、约束和非功能特性这几个角度来描述构件连接器.

定义 **1.5**. 连接器是一个六元组

$$Connector=\langle ID,Role,Beha,Msgs,Cons,Nfun\rangle.$$

其中:*ID* 是连接器的标识;*Role* 是连接器与构件的交互点的集合;*Beha* 是连接器行为的语义描述;*Msgs* 是连接器各 *Role* 中事件产生的消息的集合;*Cons* 是连接器约束的集合,包括连接器的初始条件、前置条件和后置条件;*Nfun* 是连接器的非功能说明.

Role 集合中每个交互点 $role=\langle Id,Action,Event,Lconstrains\rangle$.其中:*Id* 是 *role* 的标识;*Action* 是 *role* 活动的集合,每个活动由事件的连接(谓词)组成 *Event* 是 *Role* 产生的事件集合;*Lconstrains* 是 *Role* 的约束集合.把 *Role* 与连接器的其他属性分开来描述的目的是突出连接器的多态性,即一个连接器可以同时与多个构件相连.*Nfun* 通常包括连接器的安全性、可靠性说明等.

尽管连接器可以以构件方式实现,但它不容易进行功能封装,不具有执行自治性,仅为满足构件交互需求存在.目前,程序设计语言中对构件连接器的直接支持较少,事件模型、组装标准等都不直接隶属于程序设计语言.连接的表示与实现多为隐式,仅以抽象形式在建模中出现.构件连接器对体系结构的非功能性质量属性有重要影响.构件着重于软件的功能需求,而构件连接器与软件的非功能需求关系密切,对体系结构适应性、重用性、维护性等有很大作用.动态连接是体系结构行为适应的基础,构件连接器是处理构件交互的专门部件,通过对连接的定制、调整,可以方便地更改构件的连接关系,改变体系结构.

软件体系结构包括构件、构件之间的交互关系、约束、构件和连接器构成的拓扑结构、设计原则与指导方针,所以软件体系结构是一个设计,它包括所建立系统中各组成元素(构件和连接器)的描述、元素之间的交互、指导装配的范例和对范例的约束.体系结构是对一类具有相似组织结构的系统模式的抽象,它定义了构件、连接器类型和一系列关于它们如何组合的限制规则.

2 模型的精化

2.1 系统的精化

我们采用 RUP 迭代式实现方法,将设计过程分解为一系列小的迭代段,给定系统规范 SP,构建一个正确实现 SP 的程序 P,再逐步向原始规范中添加信息,将复杂的设计过程分解成为一个个小的易于理解与验证的迭代段:

$$SP_0 \subseteq SP_1 \subseteq \ldots \subseteq SP_n.$$

其中:P_0 是原始的需求规范,对于 $i=1,\ldots,n, SP_{i-1} \subseteq SP_i$ 为一步精化;SP_n 为最终的规范,SP_n 正确地描述了程序的所有细节,包含了所有的设计决策,是 SP 的正确实现.

如图 1 所示,体系结构的精化就是将系统所要提供的服务通过接口分派给构件,连接器精化为接口提供的服务的组合,软件系统的目的是为用户提供服务,如图 1(a)所示.在系统精化过程中,任务通过接口分配给构件,如图 1(b)所示.构件之间按照一定的体系结构风格通过连接器相连,如图 1(c)所示.过程如下所示:

- 图 1(a)中的 P_{sys} 描述系统提供的服务;
- 该服务被分配到系统的接口;
- 然后,系统接口将该任务分配到构件 C,如图 1(b)所示;
- 为了完成任务,构件 C 需要构件 A 和 B 提供的服务.

系统的规范表示为系统接口的规范,如图1(a)所示.

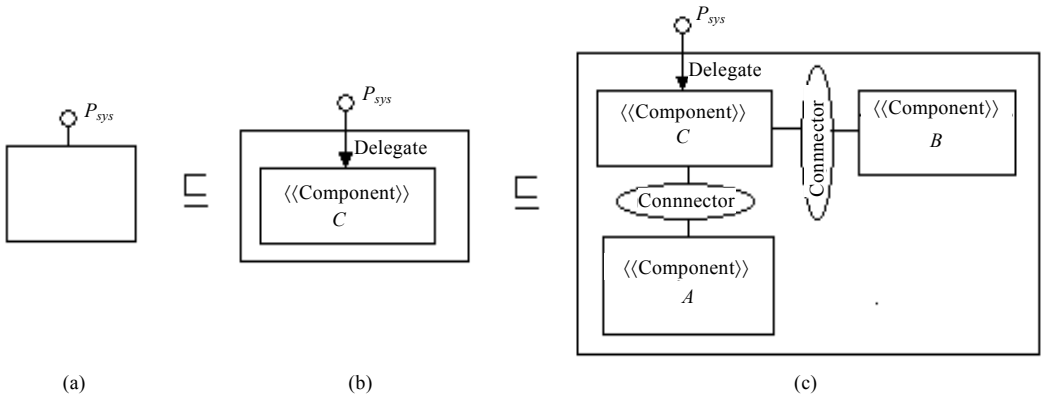

Fig.1 Refinement of software architecture

图 1 体系结构的精化

2.2 构件的精化

若构件 C 具有示出接口(provide interfaces)P 和导入接口(required interfaces)R,P 为构件 C 提供服务接口的集合 $P=\cup\{Prov_i \mid \exists i \bullet Prov_i \in intf_i.Prov_i \wedge intf_i \in Inf\}$,$R$ 表示构件 C 需要服务接口 $R=\cup\{Requ_i \mid \exists i \bullet Requ_i \in intf_i.Requ_i \wedge intf_i \in Inf\}$的集合,构件 C 的语义$[[C]]$是该构件输入服务和输出服务之间的二元关系:

2.2.1 构件语义

构件 C 的语义标识为其需要的服务与提供的服务之间的二元关系

$$[[C]](R_i, P_i') =_{df} (R_i >> C) \subseteq P_i'.$$

其中:变量 R_i 的值为需要的服务;P_i' 的值为提供的服务 $Prov$;$R_i >> C$ 代表构件 C 获得所需服务 R_i 后可提供的服务.

$$\langle Prov, F(M), A, MSpec \rangle.$$

其中:$F(M)$是将 M 中的方法除去后获得的类模型;映射 $MSpec$ 是按照给定的需要的服务通过递归方程

$MSpec(op)=I(MImpl(op))$定义的, I 表示每个调用 $op(inexp,outvar)$;$inexp$ 为实际输入参数;$outvar$ 为输出参数.

$$R_i=\langle Requ,M_R,A_R,MSpec_R\rangle.$$

注意,构件 C 若没有导入接口,则称作封闭构件(closed component).此时,$R_i>>C$ 变成常数,即封闭程序(closed program),C 的语义成为不变的常量.

在模块化程序设计范例中,构件是作为一个模块来设计和实现的,模块示出接口中的每个操作的实现程序都是过程或函数,那些过程或函数既可以在模块内部定义又可以在其他模块中定义.在此情况下,构件要调用的外部模块必须是已声明的,外部模块属性值的类型、方法的参数必须是已声明的.因此,事实上,一个构件不是一个单一的模块,而是一个包含所有声明的类型和模块的事物.在面向对象范例(如 Java)中,构件可看作实现示出接口 *Prov* 的类:

```
M;                    // the declaration of the design model
Class C implements Prov {
    attr:A;
    public:m=df MImpl(m);                  // for each m∈Prov
    private:op=df MImpl(op);}              // for each op∈Priv
```

使得 rCOS 添加接口表示法后,可以描述构件和构件精化,还可以扩展 RCOOD[16],使之可以描述基于构件的开发.

单调前向闭包:令 $intf_i\in Intf$,$\subseteq_R$ 和 $\subseteq_p$ 分别是 *Requ* 和 *Prov* 规范的精化关系.那么,

$$\subseteq_R\odot[[C]]\odot\subseteq_p=[[C]],$$

其中⊙代表关系组合.构件 C_2 是构件 C_1 的精化,表示为 $C_1\subseteq C_2$.

2.2.2 构件精化

如果 $R_1=R_2\wedge P_1=P_2\wedge[[C_2]]\Rightarrow[[C_1]]$,则构件 C_2 是 C_1 的精化.

因此,当 C_1 精化为 C_2 时,对于给定的需要的服务 *Requ*,如果 C_1 实现 *Prov*,则 C_2 通过 *Requ* 实现 *Priv*,精化的条件如下:

(1) $Dom(C_1)=Dom(C_2)$; (4) $Priv(C_2)\subseteq Priv(C_1)$; (7) $(Cons(C_2)=Cons(C_1))$ or $(Cons(C_2)\Rightarrow Cons(C_1))$;

(2) $Prov(C_1)=Prov(C_2)$; (5) $Beha(C_2)\Rightarrow Beha(C_1)$; (8) $(Nonf(C_2)=Nonf(C_1))$ or $(Nonf(C_2)\Rightarrow Nonf(C_1))$.

(3) $Requ(C_2)=Requ(C_1)$; (6) $Msgs(C_2)=Msgs(C_1)$;

2.2.3 构件组合

如果 $C_i=\langle Intf_i,M_i,A_i,MImpl_i,Init_i\rangle$为两个构件,其中 i=1,2.接口集合 Inf_1 中每个接口所提供服务的集合为 $Prov(C_1)$,Inf_2 中所提供的服务集合为 $Prov(C_2)$.C_1 和 C_2 中私有属性集合分别为 $Priv(C_1)$和 $Priv(C_2)$,需求服务的集合为 $Requ(C_1)$和 $Requ(C_2)$,假设 $Prov(C_1)\cap Prov(C_2)=\varnothing$,$Priv(C_1)\cap Priv(C_2)=\varnothing$,且 $Requ(C_1)\cap Requ(C_2)=\varnothing$.组合 $C_1||C_2$ 定义为它们的示出接口和导入接口的合并,合并后可除去两个构件中相匹配的服务*:

$$C_1||C_2=_{df}\langle O_1\cup O_2,Prov(C_1)\cup Prov(C_2),Priv(C_1)\cup Priv(C_2),Requ(C_1)\backslash Prov(C_2)\cup Requ(C_2)\backslash Prov(C_1),$$
$$Beha(C_1)\cap Beha(C_2),(Msgs(A)\cup Msgs(B)),Cons(A)\cap Cons(B),Nonf(A)\cap Nonf(B)\rangle,$$

其中,O_i 表示两个构件除去接口后的剩余部分.

此定义允许几个构件共享其他某个构件的一个示出接口,即所提供的部分服务.信息隐藏可使一个构件提供的已被另一个构件使用的服务内部化:$(C_1||C_2)\backslash(Requ(C_1)\cap Prov(C_2))\backslash(Requ(C_2)\cap Prov(C_1))$.构件的定义和精化关系反映了构件的重用特性和精化能力.

2.2.4 构件互操作

假设 C_1 和 C_2 是两个构件,如果$\exists x\in Prov(C_1)\wedge\exists y\in Requ(C_2)$使得$(x\Rightarrow y)\wedge(pre\text{-}cons(C_2))$,在此条件下,构件 C_1 通过发送消息“invocation”调用 C_2 的 $Prov(C_2)$,就执行了“invocation”演算[27,28],表示为 $Inv(C_1,C_2)$,或写成 $C_1\otimes C_2$.

* 构件合并是将相互匹配的接口连接起来,既要匹配两个接口之间语法,又要匹配语义,参见文献[29].

例如:若 C_1 和 C_2 是两个独立的构件,要将 C_1 和 C_2 连接起来,在 CORBA 中就是通过“调用”来实现的.在连接以后,C_1 和 C_2 仍然是构件,它们具有如下属性:

(1) $Dom(C_1 \otimes C_2)=Dom(C_1)\cup Dom(C_2)$;
(2) $Prov(C_1 \otimes C_2)\supseteq Prov(C_1)\cup Prov(C_2)$;
(3) $Requ(C_1 \otimes C_2)\subseteq Requ(C_1)\cup Requ(C_2)$;
(4) $Priv(C_1 \otimes C_2)\subseteq Priv(C_1)\cup Priv(C_2)$;
(5) $Beha(C_1 \otimes C_2)\Leftrightarrow Beha(C_1)\wedge Beha(C_2)$;
(6) $Msgs(C_1 \otimes C_2)=Msgs(C_1)\cup Msgs(C_2)$;
(7) $Cons(C_1 \otimes C_2)\Leftrightarrow Cons(C_1)\wedge Cons(C_2)$;
(8) $Nonf(C_1 \otimes C_2)\Leftrightarrow Nonf(C_1)\wedge Nonf(C_2)$.

2.2.5 构件的等价性

假设 C_1 和 C_2 是两个构件,如果 C_1 和 C_2 满足下列条件,则它们等价,表示为 $C_1=C_2$.

(1) $Dom(C_1)=Dom(C_2)$;
(2) $Prov(C_1)=Prov(C_2)$;
(3) $Requ(C_1)=Requ(C_2)$;
(4) $Priv(C_1)=Priv(C_2)$;
(5) $Beha(C_1)\Leftrightarrow Beha(C_2)$;
(6) $Msgs(C_1)=Msgs(C_2)$;
(7) $Cons(C_1)\Leftrightarrow Cons(C_2)$;
(8) $Nonf(C_1)=Nonf(C_2)$.

3 示例分析:网上商店管理系统

在该系统中,商店拥有各种各样的商品,每种商品有一定的库存数量.客户需要先注册才能进入商店购买商品,当客户完成订购后,商店会将商品派送给客户.

我们先分析系统所要提供的服务,然后将服务分配到不同的构件,如系统需要提供订购请求、产品入库、客户注册等服务,我们可以将那些服务分配到系统接口,系统接口再将服务委托(delegate)到构件.构件为了完成任务可能还需要其他构件提供一些服务.每个构件可以独立地设计与实现.在 UML 中,构件模型是通过类模型与交互模型实现的.

我们先建立概念体系结构模型,再精化该模型以获得具体的体系结构模型,如图 2 所示.构件接口上要提供的服务通过类模型与交互模型来实现.如构件 Store 要提供的服务是货物入库(RecordItem),即将每件商品登记到商店中.我们先建立概念模型.在 UML 中,概念模型由概念类模型 CM 和用例模型 UD 组成.

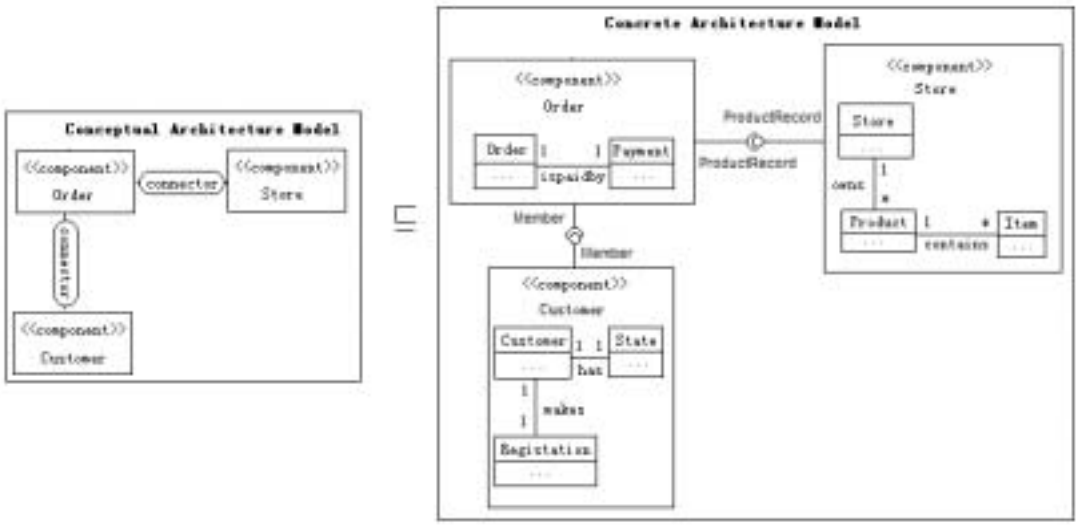

Fig.2 Refinement of the model
图 2 模型的精化

概念类模型 CM 为:

Class Store{String *name*,String *address*,String *id*};

Class Product{String *name*,String *id*};

Class Item{String *name*,String *id*};

Class Owns{Store *s*,Product *p*}; //Association classes

Class Contains{Product *p*,Item *i*};//Association classes

假定系统初始状态:ℙStore *Sto*={*s*};ℙProduct *Pro*=∅;ℙItem *It*=∅;ℙOwns *Ow*=∅;ℙContains *Co*=∅.

创建概念模型时一个重要的任务是识别用例,但用例的规范取决于概念模型的规范.在设计模型 DM 中,我们引入用例类句柄来提供用例规范的范式,以便封装概念模型.例如:用例 RecordItem 是向库中已有的商品中添加一些此商品的数量,可引入类 RecordItemHandler 来说明该用例:

用例模型 UD 为:

CM; // import the conceptual model

Class RecordItemHandler {

Method RecordItem(**val**(String *iid*,String *pid*)){

var *Item I*;

∃*p*∈*Pro*•*p.id*=*pid* ⊢

i:=**New** *Item*(*iid*);

It:=*It*∪{*i*};*Co*:=*Co*∪{⟨*p*, *i*⟩};

end *i*, *p*}};

Use Case RecordItem::{

var RecordItemHandler *h*,String *iid*,String *pid*;

h=**New** RecordItemHandler( );

read (*iid*,*pid*);

h.RecordC (*iid*,*pid*);**end** *h*,*iid*,*pid*};

主方法写作:

main( ){**var** Bool *stop*,Services *s*,*stop*:=false;

while ¬*stop* **do** {read(*s*);

if {(*s*="*RecordItem*")→RecordItem;} **fi**;

read(*stop*)};**end** *stop*,*s*}

其中:类型 Services 表示该商店管理系统提供的服务.注意:UML 概念模型还确定了一些状态不变式,如只有一个 Store 实例,我们应该检查用例以保证那些不变式成立[19].

设计模型 DM 是由设计类模型 DC 与顺序图 SD 构成的,如图 3 所示,可以通过向概念模型中添加方法等精化方式,扩充概念模型而获得.在设计阶段概念类图中的关联类 Owns 转换成 Store 与 Product 之间的方法调用,Contains 转换成 Product 与 Item 之间的调用.

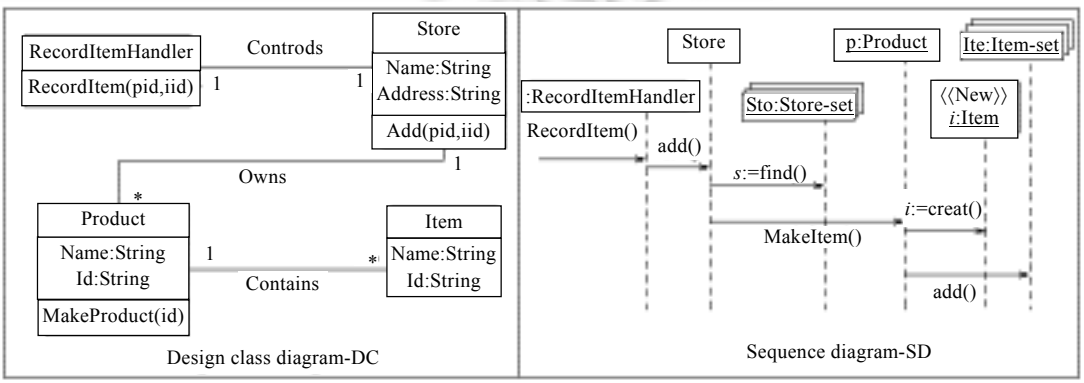

Fig.3 Design class diagram and sequence diagram

图3 设计类图和顺序图

假定对于每种类型ℙC 都声明了方法 add( ),delete( )和 find( )用来在类 C 的对象集合中进行对象的添加、

删除和查找.假设类 Pro 和 It 声明了上面列出的 3 种方法,图 3 中的设计类图和顺序图对应的程序如下:

Class Store{String *name*,String *address*;ℙProduct *Pro*; // newly added

Method add(**val** (String *cid*,String *pid*)) {**var** Product *p*; *p*=*Pro*.*find*(*pid*); *p*.*makeItem*(*iid*);**end** *p*}}

Class Publication {String *id*,String *title*,String *manufacture*,Item-Set *It*;

Method makeItem (**val** String *id*) {**var** Item *i*; *i*:=**New** Item(*id*);*It*:=*It*.*add*(*i*);**end** *i*}}

Class Item {String *id*}

Class RecordItemHandler {Product-Set *Pro*,Item-Set *It*,Contain-Set *Co*;

Method RecordItem(**val**(String *iid*,String *pid*)){*It*. add (*iid*,*pid*)}}

令 StringL 为 Store 的属性的字符列表 StringL *storeattr*≡ {String *name*×String *address*}.在设计阶段 **main**( ) 程序和概念模型中的变化不大.

main( ){**var** Bool *stop*=false,Services *s*,StringL *storeattr*,RecordItemHandler *h*,Univ *univ*;

h:=RecordItemHandler **New**();

store:=Store **New**(*storeattr*);

while ¬*stop* **do** {read(*s*); **if** {(*s*="*RecordItem*")→RecordItem} **fi**; read(*stop*)};**end** *stop*,*s*,*rch*}

构件 *Customer* 和 *Order* 可以同样用此方式来处理,构件之间通过接口相连,如图 4 所示.

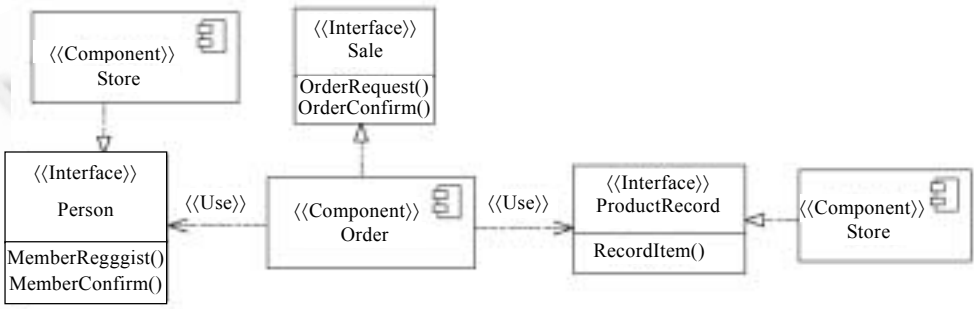

Fig.4 Components composition

图 4 构件组合

通过检查接口的语法和语义,将匹配的构件合并起来.例如,检查构件 *Store* 与 *Order*,发现 *Store* 的示出接口 *ProductRecord* 与 *Order* 的导入接口 *ProductRecord* 一致,则可将两个构件合并起来,合并后接口 *ProductRecord* 成为内部接口,从接口集合中减去.

System::=*Store*||*Order*||*Customer*.

由于我们是通过逐步精化模型使系统从抽象到具体的,所以提高了重用性与一致性.如上例中的 DM 可以通过扩展 CM 获得,所以 CM 在 DM 中得到重用,并提高了不同抽象层次模型 CM 与 DM 之间的一致性.在动态用例模型中,静态类模型 CM 是直接导入的,因此提高了动态模型与静态模型之间的一致性.另外,我们还可以通过检查状态不变式与模型良构条件来保证模型的正确性.

4 结束语

本文将形式化的建模方法应用到模型驱动式软件开发中,通过对象系统精化演算 rCOS 来提高模型描述与模型转换的精确性.按照 UML 中基于构件的开发过程构建构件、连接器和体系结构模型以及类模型与交互模型,以便利用文献[17,22]中的演算方法进行模型的转换与精化.我们分别建立系统不同视角的模型和不同抽象层次的模型,并研究了模型的统一与集成策略.将模型分成不同的抽象层次逐步求精,能够在软件演化过程中提高软件的可维护性.将形式方法与 UML 结合起来,既保持了模型的直观、方便,又提高了模型的精确性与一致性.

我们下一步的工作是:将此方法用于较大的软件系统开发,作为案例分析,并开发一套插件,使之与 UML 的

支持工具 Rational Rose 集成起来,以便提高 UML 模型的精确性,并对模型的转换与模型检查提供支持.我们还将研究多接口中的并发问题和接口匹配问题,特别是语义匹配和接口适配器的设计.

References:

[1] Mellor SJ, Balcer MJ. Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley, 2002.

[2] Hemme-Unge K, Flor T, Vögler G. Model driven architecture development approach for pervasive computing. In: Proc. of the OOPSLA 2003. New York: ACM, 2003. 314−315.

[3] Favre L. Foundations for MDA-based forward engineering. Journal of Object Technology, 2005,4(1):129−153.

[4] Frankel DS. Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Son, 2003.

[5] MDA. Specifications. 2005. http://www.omg.org/mda/specs.htm#MDAGuide

[6] Medvidovic N, Rosenblum DS, Redmiles DF, Robbins JE. Modeling software architectures in the unified modeling language. ACM Trans. on Software Engineering and Methodology, 2002,11(1):3−57.

[7] Object Management Group. Unified modeling language specification. ver2.0. 2004. http://www.omg.org/technology/documents/

[8] Garlan D, Kompanek A. Reconciling the needs of architectural description with object modeling notations. In: Evans A, Kent S, Selic B, eds. Proc. of the 3rd Int'l Conf. on the UML. LNCS 1939, Berlin: Springer-Verlag, 2000. 498−513.

[9] Giese H, Vilbig A. Separation of non-orthogonal concerns in software architecture and design. In: Choren R, Garcia A, Lucena C, , Romanovsky A, eds. Software and System Modeling (SoSyM). LNCS 3390, Berlin: Springer-Verlag, 2005. 272−289.

[10] Paige RF. Integrating a program design calculus and a subset of UML. The Computer Journal, 1999,42(2):82−99.

[11] Amálio N, Stepney S, Polack F. Formal proof from UML models. In: Davies J, *et al*. eds. Proc. of the ICFEM 2004. LNCS 3308, Berlin: Springer-Verlag, 2004. 418−433.

[12] Kim SK, Carrington D, Duke R. A metamodel-based transformation between UML and object-Z. In: Proc. of the IEEE Symp. on Human-Centric Computing Languages and Environments (HCC 2001). Stresa: IEEE Computer Society, 2001. 112−119.

[13] Kim SK, Carrington D. Using integrated metamodeling to define OO design patterns with object-Z and UML. In: Proc. of the 11th Asia-Pacific Software Engineering Conf. (APSEC 2004). Los Alamitos: IEEE Computer Society, 2004. 257−264.

[14] Liu Z, He J, Li X. rCOS: Refinement of component and object systems. In: de Boer FS, Bonsangue MM, Graf S, de Roever WP, eds. Proc. of the 3rd Int'l Symp. on Formal Methods for Components and Objects (FMCO 2004), Revised Lectures. LNCS 3657, Berlin: Springer-Verlag, 2005. 183−221.

[15] He J, Liu Z, Li X, Qin S. A relational model for object-oriented designs. In: Chin WN, ed. Proc. of the 2nd Asian Symp. on Programming Languages and Systems (APLAS 2004). LNCS 3302, Berlin: Springer-Verlag, 2004. 415−426.

[16] He J, Liu Z, Li X. Towards a refinement calculus for object-oriented systems. In: Wang Y, ed. Proc. of the ICCI 2002 as a Keynote talk. Alberta: IEEE Computer Society, 2002. 69−76.

[17] Liu Z, He J, Liu J, Li X. Unifying views of UML. Electronic Notes in Theoretical Computer Science, 2004,101C:95−127.

[18] Liu Z, He J, Li X, Chen Y. A relational model for formal object-oriented requirement analysis in UML. In: Dong JS, Woodcock J, eds. Proc. of the ICFEM 2003. LNCS 2885, Berlin: Springer-Verlag, 2003. 641−661.

[19] Liu J, Liu Z, He J, Li X. Linking UML models of design and requirement. In: Strooper P, ed. Proc. of the Australian Software Engineering Conf. Melboune: IEEE Computer Society Press, 2004. 329−338.

[20] Li X, Liu Z, He J. Consistency checking of UML requirements. In: Shafer DF, ed. Proc. of the ICECCS 2005. Los Alamitos: IEEE Computer Society Press, 2005. 411−420.

[21] Liu J, Miao H, Gao X. A specification-based software construction framework for reuse. In: George C, Miao H, eds. Proc of the ICFEM2002. LNCS 2495, Berlin: Springer-Verlag, 2002. 69−79.

[22] Hoare C, He J. Unifying Theories of Programming. Prentice-Hall International, 1998.

[23] Liu Z, Li X, He J. Using transition system to unify UML models. In: George C, Miao H, eds. Proc. of the ICFEM 2002. LNCS 2495, Berlin: Springer-Verlag, 2002. 535−547.

[24] Luckham DC, Vera J. An event-based architecture definition language. IEEE Trans. on Software Engneering, 1995,21(9):717−734.

[25] Medvidovic N, Taylor RN. A classification and comparison framework for software architecture description languages. IEEE Trans. on Software Engineering, 2000,26(1):70−93.

[26] 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

[27] Taylor RN, Medvidovic N, Anderson KM, Dubrow DL. A component and message-based architectural style for GUI software. IEEE Trans. on Software Engineeting, 1996,22(6):390−406.

[28] Zhao HQ,Wang GR. An abstract model of software architecture. Chinese Journal of Computers, 2002,25(7):730−736 (in Chinese with English abstract).

[29] Liu Z, He J, Li X. Contract oriented development of component software. IFIP WCC-TCS. Toulouse: Kluwer, 2004. 355−372.

附中文参考文献:

[26] 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

[28] 赵会群,王国仁.软件体系结构抽象模型.计算机学报,2002,25(7):730−736.

刘静(1965－),女,江苏徐州人,博士,副教授,主要研究领域为软件工程,模型驱动式软件开发,构件技术,形式方法.

何积丰(1943－),男,教授,博士生导师,主要研究领域为形式方法,软件工程,高可信软件.

缪淮扣(1953－),男,教授,博士生导师,主要研究领域为软件工程,形式方法,软件建模.