支持设计时重用软件的反射式软件体系结构及PMB协议研究

罗巨波[1] 应 时[2]

(重庆理工大学计算机科学与工程学院 重庆 400054)[1] (武汉大学软件工程国家重点实验室 武汉 430072)[2]

摘 要 给出了支持软件体系结构设计时重用的反射式软件体系结构,描述了反射式软件体系结构的元级软件体系结构和基级软件体系结构之间进行交互和互操作的协议PMB,基于软件规格语言Object-Z对PMB协议进行了形式化描述。

关键词 软件体系结构重用,反射式软件体系结构,元级,基级,PMB协议

中图法分类号 TP311.5 **文献标识码** A

Study of Reflective Software Architecture and PMB Protocols for Reusing Software at Design Stage

LUO Ju-bo[1] YING Shi[2]

(College of Computer Science and Engineering, Chongqing University of Technology, Chongqing 400054, China)[1]

(State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China)[2]

Abstract This paper proposed a reflective software architecture supporting the reuse of architectural level designs, and described the PMB protocols used to complete interaction and interoperation between meta-level architecture and base-level architecture of the reflective software architecture. Finally, it formalized the PMB protocols using the formal specification language, Object-Z language.

Keywords Reuse of software architecture, Reflective software architecture, Meta-level, Base-level, PMB protocols

1 引言

虽然特定领域的软件体系结构[1]、软件体系结构模式[2]、软件体系结构风格[3]和体系结构框架[4]等在一定程度上,可以帮助人们重用软件体系结构,但是这些重用方法也存在一些问题,主要包括:缺乏统一的体系结构建模方法、基于不同的体系结构描述语言和缺乏支持重用过程的信息。

文献[5]将元信息、元建模、反射和软件体系结构结合起来,构造了一种在设计阶段支持软件体系结构重用的反射机制RMRSA。重用对象是体系结构层的组件、连接器、体系结构片段、风格,是一种更通用、更便捷的体系结构制品本身的重用方法,它具有统一的体系结构建模方法的信息,给出了统一的元级体系结构描述语言MetaADL,可利用反射机制来屏蔽基级体系结构描述语言,以实现体系结构设计时制品的重用。

本文将在此基础上做如下工作:完善反射式软件体系结构的设计;部分定义反射式软件体系结构的元级体系结构和基级体系结构之间的交互和互操作PMB协议;为了使PMB协议达到一定的正确性和精确性,基于软件规格语言Object-Z对其进行形式化描述。

2 反射式软件体系结构

体系结构的反射是指:利用元级体系结构中关于基本级体系结构语义和用法等元信息,对基本级体系结构的使用进行管理和控制的机制。我们把基于反射机制的软件体系结构称作反射式软件体系结构。

反射式软件体系结构由基本级体系结构、元级体系结构和PMB(Protocol for connecting Meta-level architecture and Base-level architecture)协议3部分组成,如图1所示。通过定义抽取基级体系结构的重用元信息的具体化操作和利用元级体系结构元信息描述来获得基级体系结构的反射操作,体系结构设计人员和工具可以在元级对体系结构重用元信息这一高抽象层次设计元素进行组合与重用,从而达到重用已有的体系结构设计结构,构造满足需求的软件体系结构的目的。反射式软件体系结构的元级体系结构为基本级体系结构提供一种封装机制,并为设计人员或设计工具提供一种使用基本级体系结构的标准接口。那么,基于PMB协议,元级体系结构和基本级体系结构一起可以使软件体系结构成为一种具有自包含和自描述特性的、可重用的大粒度软件组件。

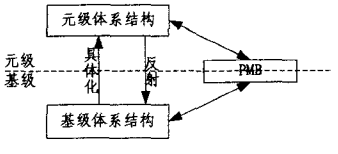

图1 反射式软件体系结构

反射式软件体系结构的基级体系结构采用某种特定ADL进行描述。本文以C2SADEL为案例来进行说明。

到稿日期:2009-11-18 返修日期:2010-02-01 本文受国家自然科学基金项目(60473066)资助。

罗巨波(1976—),男,博士,讲师,主要研究方向为软件体系结构和模式、软件复用,E-mail:whuluocheng@126.com;应 时(1965—),男,博士,教授,博士生导师,主要研究方向为基于组件的软件工程方法学、软件体系结构和模式、软件的可重用性和互操作性等。

C2SADEL 的体系结构模型分为 3 大部分：组件类型定义部分 component_types：定义了一组组件类型；连接器类型定义部分 connector_types：定义了一组连接器类型；拓扑结构定义部分 archietectural_topology：定义了一组组件 component_instances、一组连接器 connector_instances 以及这些组件与连接器之间的连接关系 connections。

基级体系结构元素除了 C2SADEL 中定义的数据外，还定义了一组操作，以支持反射机制对基级的修改。所有的基级元素(component_instance，connector_instance，archietectural_topology，connection)都有一个 update 方法，可以被 PMB 调用，对基级元素按照元级体系结构定义的元信息进行更新，同时各种元素都定义了一系列对应的读取、添加、删除方法以支持对基级体系结构的修改。

反射式软件体系结构的元级体系结构是使用 MetaADL 描述基级体系结构中的元信息构造出来的。元组件封装并描述基级中相应组件及其类型的各种元信息。元连接器封装并描述基级中相应连接器及其类型的各种元信息。元组合件封装并描述基级中体系结构本身的元信息，包含体系结构的外观元信息、构成元信息和配置元信息。

在元级体系结构中，所有的元级体系结构元素都是 ElementofMetaLevel 的子类，因此都有标识符 Identifier、对应的基级体系结构元素 elementofBaseLevel、可变性 changeable 3 个基本属性；同时都有 getData 操作得到的具体的元信息数据，有 notify 操作当该元素发生变化时通知基级体系结构，有 verify 操作可以检查重用的约束条件是否满足。MetaComponent，MetaConnector 和 MetaComposite 3 个元素是体系结构(组合件)的构成元素 ArchitecturalConstituent 的子类，都有基本特征 basicFeature、结构 structure、一组行为 behavior、一组约束 constraint 和一组属性 property，同时具有这些子元素的读取、添加、删除和修改方法。MetaComposite 除了具有将自身作为体系结构构成元素所需要定义的外观元信息外，还包括内部的构成元信息 constituent 和配置元信息 configuration。同时提供了对这些元信息的读取、添加、删除和修改方法。

元级体系结构中建立的组件、连接器和组合件中定义的元信息是对基级体系结构的具体描述。元级体系结构组件、连接器和组合件中定义的操作由 PMB 协议调用执行，以完成元级体系结构的更新。

3 PMB 协议

PMB 协议用于支持元级体系结构和基级体系结构之间的交互和互操作，包括反射操作和具体化操作，这两种操作都为支持软件重用的操作服务。支持软件重用的操作包括：增加组件操作、删除组件操作、增加连接器操作、删除连接器操作、增加链接操作、删除链接操作。任何一个重用操作，首先需要从元级获取有关的元信息；然后根据所执行的操作，修改元级体系结构；最后根据修改后的元级体系结构，反射生成基级体系结构，使元级始终正确地表示当前基级体系结构的元信息。

PMB 协议为各种重用操作定义其详细的操作规程，这些操作规程描述元级体系结构和基级体系结构之间的交互和互操作过程。操作规程保证了元级体系结构在改变的同时改变基级结构，基级体系结构的改变也能在元级软件体系结构中得到相应的改变，确保了元级与基级之间的因果关联始终是正确的。为了保证反射式软件体系结构的基和本级体系结构元级体系结构之间的交互和互操作的正确性、完整性和精确性，我们运用软件规格语言 Object-Z 来描述 PMB 协议。

经过任何一个或几个重用操作后，反射式体系结构的元级和基级的分量会发生变化，接着我们对这些会发生变化的分量进行分析和说明。

list_metacomponent 表示元组件的列表，list_metaconnector 表示元连接器的列表，list_component_instances 表示组件的列表，list_connector_instances 表示连接器的列表。

组件和连接器连接的关系 ConnectionComConn：Component_instance ⊢→Connector_instance 定义了从组件 Component_instance 到连接器 Connector_instance 的映射关系，Component_instances 是其定义域，Connector_instances 是其值域。连接器和连接器连接的关系 ConnectionConnConn：Connector_instance ⊢→Connector_instance 定义了从连接器 Connector_instance 到连接器 Connector_instance 的映射关系，connector_top_instances 是其定义域，connector_down_instances 是其值域。

元组件和元连接器链接的关系 LinkComConn：MetaComponent ⊢→MetaConnector 定义了元组件 MetaComponent 到元连接器 MetaConnector 的映射关系，metacomponents 是其定义域，metaconnectors 是其值域。元连接器和元连接器连接的关系 LinkConnConn：MetaConnector ⊢→MetaConnector 定义了从元连接器 MetaConnector 到元连接器 MetaConnector 的映射关系，metaconns_top 是其定义域，metaconns_down 是其值域。

list_metacomponent 是表示元组件到与其具有因果关联的组件的映射关系 MetacomToInstance：MetaComponent↔Component_instance 的定义域，list_component_instances 是其值域。

list_metaconnector 是表示元连接器到与其具有因果关联的连接器的映射关系 MetaconnToInstance：MetaConnector↔Connector_instance 的定义域，list_connector_instances 是其值域。

有了上述解释，下面以增加组件操作为代表，详细给出增加组件重用操作的操作规程及其基于软件规格语言 Object-Z 的形式化描述。

3.1 增加组件操作的操作规程

增加组件操作 addComponent，将一个待重用的组件添加到软件体系结构中。

增加组件操作 addComponent 的详细操作规程定义如下：

1. PMB 接收 COMPOSITE_ADDCOMPONENT 事件。

2. PMB 调用元级体系结构元组合 MetaComposite 的 addConstituentElement() 方法来添加元组件 newMetaComponent。

3. 根据将要添加元组件 newMetaComponent 的 identifier，PMB 调用元组合件 MetaComposite 的配置元信息 ConfigurationMetaInfo 的 LinkMetaInfo 分量，遍历元级体系结构所有的链接二元组(SourcePointMetaInfo，TargetPointMe-

taInfo)，查找待增加的元组件 newMetaComponent 将要插入的位置。

4. 如果遍历元级体系结构所有的链接二元组(SourcePointMetaInfo，TargetPointMetaInfo)都没有找到合适的二元组，则表示待增加的元组件 newMetaComponent 将要插入的位置是在元级体系结构的叶子端，也即 newMetaComponent 直接和元级体系结构的某一个符合条件的元连接器 MetaConnector01 相连，则按以下步骤(1)到步骤(8)进行操作(操作的辅助图示说明见图 2)。

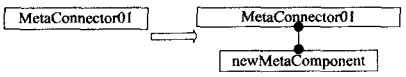

图 2　插入组件到反射式体系结构的叶子端

(1)调用元组合件 MetaComposite 的配置元信息类 ConfigurationMetaInfo 的链接元信息类 LinkMetaInfo 的方法 addlinkMetaInfo()增加链接二元组(newMetaComponent，MetaConnector01)；根据元组合件 MetaComposite 的构成约束元信息和配置约束元信息，调用元组合件的 verify 方法检查添加链接二元组(newMetaComponent，MetaConnector01)后约束条件是否满足。

(2)根据元连接器 MetaConnector01 和元组件 newMetaComponent 的约束元信息 ConnectorConstraintMetaInfo 和 ComponentConstraintMetaInfo，分别调用 ConnectorMetaInfo 和 ComponentMetaInfo 的 verify()方法检查添加新的元组件 newMetaComponent 后约束条件是否满足。

(3)检查步骤(1)和步骤(2)中的约束条件是否满足，只要有任何一个约束条件不满足，则进行以下操作：(a)撤销添加新的元组件 newMetaComponent 和元连接器 MetaConnector01 操作。(b)撤销添加链接二元组(newMetaComponent，MetaConnector01)；(c)转向步骤(6)。如果检查步骤(1)和步骤(2)中的约束条件全部满足，则继续下一步的操作。

(4)调用组件元信息列表类 ComponentMetaInfoList 的 addComponentMetaInfo()方法在 ComponentMetaInfoList 中增加一个元组件 newMetaComponent。

(5)PMB 调用元组合件 MetaComposite 的 notify()方法，通知基级体系结构进行更新。

(6)基级体系结构调用元组合件 MetaComposite 的 getConstituentElement()方法得到元组件 newMetaComponent，再调用 Architectural_topology 的 addComponentInstance()方法添加组件 newComponent。

(7)基级体系结构调用元组合件 MetaComposite 的 getConfiguration()方法得到配置元信息：二元组(newMetaComponent，MetaConnector01)，根据得到的配置元信息，接着基级体系结构调用 Architectural_topology 的 addConnection()方法添加一个连接(newComponent，Connector01)。

(8)增加组件 addComponent，操作规程结束。

5. 如果遍历元级体系结构所有的链接二元组(SourcePointMetaInfo，TargetPointMetaInfo)找到了合适的二元组(MetaComponent02，MetaConnector02)，则表示待增加的元组件 newMetaComponent 将要插入的位置是在元级体系结构的枝干部分，则按以下步骤(9)到步骤(19)进行操作(操作的辅助图示见图 3)。

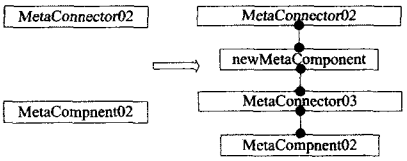

图 3　插入组件到反射式体系结构的枝干部分

(9)调用元组合件 MetaComposite 的配置元信息类 ConfigurationMetaInfo 的链接元信息 LinkMetaInfo 的方法 deletelinkMetaInfo()，删除链接二元组(MetaComponent02，MetaConnector02)。根据元连接器 MetaConnector01 和元组件 newMetaComponent 的约束元信息 ConnectorConstraintMetaInfo 和 ComponentConstraintMetaInfo，分别调用 ConnectorMetaInfo 和 ComponentMetaInfo 的 verify()方法检查添加新的元组件 newMetaComponent 和元连接器 MetaConnector01 后约束条件是否满足。

(10)新增一个元连接器 MetaConnector03，根据元连接器 MetaConnector03 的约束元信息 ConnectorConstraintMetaInfo 和元组件 newMetaComponent 的约束元信息 ComponentConstraintMetaInfo，分别调用 ConnectorMetaInfo 和 ComponentMetaInfo 的 verify()方法检查约束条件是否满足。

(11)调用元组合件 MetaComposite 的配置元信息类 ConfigurationMetaInfo 的链接元信息 LinkMetaInfo 的 addlinkMetaInfo()方法增加 3 个二元组(MetaComponent02，MetaConnector03)，(newMetaComponent，MetaConnector03)和(newMetaComponent，MetaConnector02)链接。

(12)根据元组合件 MetaComposite 的构成约束元信息和配置约束元信息，调用元组合件 MetaComposite 的 verify()方法检查约束条件是否满足；如果不满足，则删除该元组件和新增加的 3 个链接二元组，并转向步骤(6)；否则继续第(13)步的操作。

(13)检查步骤(9)，步骤(10)和步骤(12)中的约束条件是否满足，只要有任何一个约束条件不满足，则进行以下操作：

(a)撤销断开二元组(MetaComponent02，MetaConnector02)链接操作；(b)撤销增加元连接器 MetaConnector03 和元组件 newMetaComponent 操作；(c)撤销增加 3 个二元组(MetaComponent02，MetaConnector03)，(MetaConnector03，newMetaComponent)，(newMetaComponent，MetaConnector02)链接操作；(d)转向步骤(6)。

如果检查步骤(9)，步骤(10)和步骤(12)中的约束条件全部满足，则继续下一步的操作。

(14)调用组件元信息列表 ComponentMetaInfoList 的 addComponentMeta()方法在 ComponentMetaInfoList 中增加一个元组件 newMetaComponent。

(15)调用连接器元信息列表 ConnectorMetaInfoList 的 addConnectorMeta()方法在 ConnectorMetaInfoList 中增加一个元连接器 MetaConnector03。

(16)协调器调用元组合件 MetaComposite 的 notify()方法，通知基级体系结构进行更新。

(17)基级体系结构调用元组合件 MetaComposite 的 getConstituentElement()方法得到元组件 newMetaComponent，再调用 Architectural_toplogy 的 addComponentInstance()方

法添加一个组件 newComponent。

(18) 基级体系结构调用元组合件 MetaComposite 的 getConstituentElement() 方法得到元连接器 MetaConnector03，再调用元连接器 MetaConnector03 的 getData()方法得到具体的元信息数据；再调用 Architectural_toplogy 的 addConnectorInstance()方法添加一个连接器 Connector03。

(19)基级体系结构调用元组合件 MetaComposite 的 getConfiguration()方法得到配置元信息：3 个二元组(MetaComponent02，MetaConnector03)，(MetaConnector03，newMetaComponent)，(newMetaComponent，MetaConnector02)，根据得到的配置元信息，接着基级体系结构调用 Architectural_toplogy 的 addConnection()方法添加 3 个连接(Component02，Connector03)，(Connector03，new Component)，(newComponent，Connector02)。

6. 增加组件 addComponent，操作规程结束。

3.2 增加组件操作的形式化描述

根据 3.1 小节对 addComponent 操作规程的定义，如果待增加的元组件 newMetaComponent 将要插入的位置是在元级体系结构的叶子端，我们定义其操作形式化描述的前置条件模式为 Pre_addComponent_One，操作模式为 addComponent_One。

前置条件模式 Pre_addComponent_One 的定义如下：

Pre_addComponent_One

$ElementofMetaLevel$
$metacomposite : MetaComposite$
$MetaConnector01 : MetaConnector$
$newMetaComponent? : MetaComponent$

$newMetaComponent?.verify(newMetaComponent?.metacomconstra\,int.Constra\,int\,Expression) = true \wedge$
$MetaConnector01.verify(MetaConnector01.metaconnconstra\,int.Constra\,int\,Expression) = true \wedge$
$metacomposite.verify(metacomposite.metacompoconstituent.metacompoconsticonstr.Constra\,int\,Expression) = true \wedge$
$metacomposite.verify(metacomposite.metacompoconfig.configconstra\,int.Constra\,int\,Expression) = true$

操作模式 addComponent_One 的定义如下：

addComponent_one

$\Delta Re\,flectiveArchitecture$
$metacomposite : MetaComposite$
$MetaConnector01 : MetaConnector; Connector01 : Connector\_ins\,tance$
$newMetaComponent? : MetaComponent; newComponent : Component\_ins\,tance$
$MetaComToIns\,tance$
$MetaConnToIns\,tance$

$newMetaComponent?.verify(newMetaComponent?.metacomconstra\,int.Constra\,int\,Expression) = true \wedge$
$MetaConnector01.verify(MetaConnector01.metaconnconstra\,int.Constra\,int\,Expression) = true \wedge$
$metacomposite.verify(metacomposite.metacompoconstituent.metacompoconsticonstr.Constra\,int\,Expression) = true \wedge$
$metacomposite.verify(metacomposite.metacompoconfig.configconstra\,int.Constra\,int\,Expression) = true$
$list\_metacomponent' = list\_metacomponent \cup \{newMetaComponent?\}$
$list\_metaconnector' = list\_metaconnector$
$metacomponents' = metacomponents$
$metaconnectors' = metaconnector$
$metaconns\_top' = metaconns\_top$
$metaconns\_down' = metaconns\_down$
$LinkComConn' = LinkComConn$
$LinkComConn' = LinkComConn \cup \{newMetaComponent? \mapsto MetaConnector01\}$
$newComponent = architectural\_topo\,log\,y.addComponentIns\,tance(newMetaComponent?)$
$component\_ins\,tances' = component\_ins\,tances \cup \{newComponent\}$
$ConnectioncomconnConnectioncomconn$
$connector\_top\_ins\,tances' = connector\_top\_ins\,tances$
$connector\_down\_ins\,tances' = connector\_down\_ins\,tances$
$ConnectioncomconnConnectioncomconn \cup \{newComponent \mapsto Connector01\}$
$component\_ins\,tances' = component\_ins\,tances$
$connector\_ins\,tances' = connector\_ins\,tances$
$list\_component\_ins\,tances' = list\_component\_ins\,tances \cup \{newComponent\}$
$list\_connector\_ins\,tances' = list\_connector\_ins\,tances$
$MetaComToIns\,tance' = MetaComToIns\,tance \cup \{newMetaComponent? \mapsto newComponent\}$
$MetaConnToIns\,tance' = MetaConnToIns\,tance$
$newMetaComponent?.metacomstructure.ComponentName = 'meta' + newComponent.name$
$newMetaComponent?.metacomstructure.TypeName = newComponent.component\_type$
$newMetaComponent?.metacomstructure.metacominterface$
$= newMetaComponent?.metacomstructure.setInterface(newComponent.component\_type.interface\_element)$
$newMetaComponent?.metacomstructure.Changeable$
$= newComponent.componentstructure.Changeable$
$newMetaComponent?.metacomponentbehavior.Changeable$
$= newComponent.componentbehavior.Changeable$
$newMetaComponent?.metacomponentbehavior.metacombehaviorspec$
$= newMetaComponent?.metacomponentbehavior.setSpecification(newComponent.component\_type.operation)$
$newMetaComponent?.metacomconstraint$
$= newMetaComponent?.setConstraint(newComponent.component\_type.invariant)$
$newMetaComponent?.metacomproperty$
$= newMetaComponent?.setProperty(newComponent.component\_type.state)$

根据 3.1 小节对 addComponent 操作规程的定义，如果待增加的元组件 newMetaComponent 将要插入的位置是在元级体系结构的枝干部分，我们定义其操作的形式化描述的前置条件模式为 Pre_addComponent_Two，操作模式为 addComponent_Two。

前置条件模式 Pre_addComponent_Two 的定义如下：

Pre_addComponent_Two

$ElementofMetaLevel$
$metacomposite : MetaComposite$
$MetaConnector03 : MetaConnector$
$newMetaComponent? : MetaComponent$

$newMetaComponent?.verify(newMetaComponent?.metacomconstra\,int.Constra\,int\,Expression) = true \wedge$
$MetaConnector03?.verify(MetaConnector03?.metaconnconstra\,int.Constra\,int\,Expression) = true \wedge$
$metacomposite.verify(metacomposite.metacompoconstituent.metacompoconsticonstr.Constra\,int\,Expression) = true \wedge$
$metacomposite.verify(metacomposite.metacompoconfig.configconstra\,int.Constra\,int\,Expression) = true$

操作模式 addComponent_Two 的定义如下：

addComponent_Two

Declarations

Predicates

其中，操作模式 addComponent_Two 的声明部分定义如下：

$\Delta Re\,flectiveArchitecture$
$metacomposite : MetaComposite$
$MetaConnector03? : MetaConnector$
$newMetaComponent? : MetaComponent$
$Connector03 : Connector\_ins\,tance$
$newComponent : Component\_ins\,tance$

操作模式 addComponent_Two 的谓词部分定义如下：

$newMetaComponent?.verify(newMetaComponent?.metacomconstra\,int.Constra\,int\,Expression) = true \wedge$
$MetaConnector03?.verify(MetaConnector03?.metaconnconstra\,int.Constra\,int\,Expression) = true \wedge$
$metacomposite.verify(metacomposite.metacompoconstituent.metacompoconsticonstr.Constra\,int\,Expression) = true \wedge$
$metacomposite.verify(metacomposite.metacompoconfig.configconstra\,int.Constra\,int\,Expression) = true$
$list\_metacomponent' = list\_metacomponent \cup \{newMetaComponent?\}$
$list\_metaconnector' = list\_metaconnector \cup \{MetaConnector03?\}$
$metacomponents' = metacomponents \cup \{newMetaComponent?\}$
$metaconnectors' = metaconnectors \cup \{MetaConnector03?\}$
$metaconns\_top' = metaconns\_top$
$metaconns\_down' = metaconns\_down$
$LinkConnConn' = LinkConnConn$
$LinkComConn' = \{LinkComConn \setminus \{MetaComponent02 \mapsto MetaConnector02\}\} \cup \{newMetaComponent? \mapsto MetaConnector03?, newMetaComponent? \mapsto MetaConnector02, MetaComponent02 \mapsto MetaConnector03?\}$
$newComponent = architectural\_topo\,log\,y.addComponentIns\,tance(newMetaComponent?))$
$Connector03 = architectural\_topo\,log\,y.addConnectorIns\,tance(MetaConnector03?))$
$component\_ins\,tances' = component\_ins\,tances \cup \{newComponent\}$
$connector\_ins\,tances' = connector\_ins\,tances \cup \{connector03\}$
$Connectioncomconn' = \{Connectioncomconn \setminus \{Component02 \mapsto Connector02\}\} \cup \{newComponent \mapsto Connector03, newComponent \mapsto Connector02, Component02 \mapsto Connector03\}$
$ConnectioncomconnConnectioncomconn$
$connector\_top\_ins\,tances' = connector\_top\_ins\,tances$
$connector\_down\_ins\,tances' = connector\_down\_ins\,tances$
$list\_component\_ins\,tances' = list\_component\_ins\,tances \cup \{newComponent\}$
$list\_connector\_ins\,tances' = list\_connector\_ins\,tances \cup \{connector03\}$
$MetaComToIns\,tance' = MetaComToIns\,tance \cup \{newMetaComponent? \mapsto newComponent\}$
$newMetaComponent?.metacomstructure.ComponentName = 'meta' + newComponent.name$
$newMetaComponent?.metacomstructure.TypeName = newComponent.component\_type$
$newMetaComponent?.metacomstructure.metacominterface$
$= newMetaComponent?.metacomstructure.setInterface(newComponent.component\_type.interface\_element)$
$newMetaComponent?.metacomstructure.Changeable$
$= newComponent.componentstructure.Changeable$
$newMetaComponent?.metacomponentbehavior.Changeable$
$= newComponent.componentbehavior.Changeable$
$newMetaComponent?.metacomponentbehavior.metacombehaviorspec$
$= newMetaComponent?.metacomponentbehavior.setSpecification(newComponent.component\_type.operation)$
$newMetaComponent?.metacomconstraint$
$= newMetaComponent?.setConstraint(newComponent.component\_type.invariant)$
$newMetaComponent?.metacomproperty$
$= newMetaComponent?.setProperty(newComponent.component\_type.state)$

$MetaConnToIns\,tance' = MetaConnToIns\,tance \cup \{MetaConnector03? \mapsto connector03\}$
$MetaConnector03?.message\_filters$
$= MetaConnector03?.setFilteringMode(connector03.connector\_type.message\_filters)$
$MetaConnector03?.metaconnstructure.ConnectorName = 'meta' + connector\_ins\,tance.name$
$MetaConnector03?.metaconnstructure.TypeName = connector03.connector\_type$
$MetaConnector03?.metaconnstructure.metaconninterface$
$= MetaConnector03?.metaconnstructure.setInterface$

(connector03.connector _type.interface _element)
MetaConnector03?.metaconnstructure.Changeable
= connector03.metaconnstructure.Changeable
MetaConnector03?.connectorbehavior.Changeable
= connector03.connectorbehavior.Changeable
MetaConnector03?.metaconnbehavior.SpecificationMetaInfo
= MetaConnector03?.metaconnectorbehavior.setSpecification
(connector03.connector _type.operation)
MetaConnector03?.metaconnconstraint
= MetaConnector03?.setConstraint(connector03.connector _type.invariant)
MetaConnector03?.metaconnproperty
= MetaConnector03?.setProperty(connector03.connector _type.state)

所以，定义增加组件操作的完整性描述模式 T_AddComponent 为：

$$T\_addComponent \triangleq (addComponent\_One \wedge Success) \vee (addComponent\_Two \wedge Success)$$

4 相关工作

对于在软件设计阶段软件体系结构的重用，国内外研究机构提出了很多不同的方法，目前具有代表意义的研究成果和本文提出的基于反射式软件体系结构软件重用的对比研究有：面向领域的体系结构重用需要针对特定领域，而本文的软件体系结构重用方法具有通用性；体系结构设计知识的重用仍然面临着许多重大的技术障碍；软件框架可以将体系结构的设计方案连同其实现代码一起进行重用是支持实现阶段而非设计阶段的重用，本文基于反射式软件体系结构的软件重用是针对软件设计阶段。

本文提出的基于反射式软件体系结构软件重用是一种更通用、更便捷的体系结构制品本身的重用方法，它具有统一的体系结构建模方法的信息，分别利用反射机制和 PMB 协议来屏蔽和完成设计阶段软件体系结构制品的重用，所以本文以及后续的研究内容在一定程度上可以部分解决软件体系结构重用存在的主要问题。

结束语 本文的主要贡献在于：给出了支持软件体系结构设计时重用的反射式软件体系结构，描述了反射式软件体系结构的元级软件体系结构和基级软件体系结构之间进行交互和互操作的协议 PMB，基于软件规格语言 Object-Z 对 PMB 协议进行了形式化描述。

我们的研究工作与已有研究不同的地方在于：(1)提供了一种在设计阶段支持软件体系结构重用的反射式软件体系结构。(2)对反射式体系结构的 PMB 协议做了部分定义。

作为今后的工作，我们将对反射式软件体系结构的元信息模型不断丰富和完善；完善 PMB 协议的定义；给出反射式软件体系结构的元级体系结构和基本级软件体系结构的一致性性质的定义；给出经过重用操作后，反射式软件体系结构的元级和基本级软件体系结构的一致性性质的证明方法和过程。

参 考 文 献

[1] Binns P, Engelhart, Vestal M. Domain-Specific Software Architectures for Guidance, Navigation, and Control [J]. Software Eng. and Knowledge Eng., 1996, 6(2): 1011-1017

[2] Shaw M. Some Patterns for Software Architecture, Pattern Languages of Program Design [M] // Vlissides, Coplien, Kerth, et al., eds. 1996: 255-270

[3] Schmerl B, Garlan D. AcmeStudio: Supporting Style-Centered Architecture Development [C] // Proceedings of International Conference on Software Engineering. Edinburgh, Scotland, May 2004

[4] Froehlich G, Hoover H J, Liu Ling, et al. Designing object-oriented frameworks, In CRC Handbook of Object Technology[M]. CRC Press, 1998

[5] 罗巨波，应时. 一种支持软件体系结构重用的反射机制及其形式化[J]. 计算机科学，2009，36(8)：145-148

(上接第 151 页)

[17] Coppolino L, Romano L, Mazzocca N, et al. Web Services workflow Reliability Estimation Through Reliability Patterns[C] // SecureComm. 2007

[18] Abramowicz W, Kaczmarek M, Zyskowski D. Duality in Web Services Reliability[C] // AICT/ICIW. 2006: 165

[19] Zhang Jia, Zhang Liang-Jie. Criteria Analysis and Validation of the Reliability of Web Services-oriented Systems[C] // ICWS. 2005: 621-628

[20] Maximilien E M, Singh M P. Toward autonomic web services trust and selection[C] // ICSOC. 2004: 212-221

[21] Maximilien E M, Singh M P. A Framework and Ontology for Dynamic Web Services Selection[J]. IEEE Internet Computing, 2004, 8(5): 84-93

[22] Liu Chuanchang, Peng Yong, Chen Junliang. Web Services Description Ontology-Based Service Discovery Model [C] // Web Intelligence. 2006: 633-636

[23] Dobson G, Lock R, Sommerville I. QoSOnt: a QoS Ontology for Service-Centric Systems [C] // EUROMICRO-SEAA. 2005: 80-87

[24] Dobson G. OWL and OWL-S for Dependability-Explicit Service-Centric Computing [C] // Service-Oriented Computing: Consequences for Engineering Requirements, SOCCER. 2006: 4

[25] Tondello G F, Siqueira F. The QoS-MO ontology for semantic QoS modeling[C] // SAC. 2008: 2336-2340

[26] Vuong Xuan Tran. WS-QoSOnto: A QoS Ontology for Web Services[C] // SOSE. 2008: 233-238

[27] Vuong Xuan Tran, Hidekazu Tsuji, Masuda R. A new QoS ontology and its QoS-based ranking algorithm for Web services [J]. Simulation Modelling Practice and Theory, 2009, 17(8): 1378-1398

[28] Onchaga R, Widya I, Morales J, et al. An ontology framework for quality of geographical information services[C] // GIS. 2008: 64

[29] Liu Gaoyong, Wang Huiling. A Method for Evaluating the Service Quality of E-commerce Websites Based on Ontology[C] // ICIII '08. 2008: 293-297

[30] Zhou Jiehan, Niemel E. Ontology-based software reliability mo-deling[C] // Software and Services Variability Management-Concepts, Models and Tools. 2007

[31] Evesti A. Quality oriented software architecture development [C] // Espoo. VTT Publications 636, 2007: 79

[32] ISO/IEC. ISO/IEC 9126-1 International Standard: Software engineering. Product quality[S]. Part 1: Quality model, 2001: 25

[33] ISO/IEC. ISO/IEC 9126-2 Technical Report: Software engineering. Product quality[S]. Part 2: External metrics, 2003: 86

[34] ISO/IEC. ISO/IEC 9126-3 Technical Report: Software engineering. Product quality[S]. Part 3: Internal metrics, 2003: 62

[35] ISO/IEC (2004) ISO/IEC 9126-4 Technical Report: Software engineering. Product quality[S]. Part 4: Quality in use metric, 2004: 59