反射式软件体系结构一致性研究

罗巨波[1,2] 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。基于Object-Z形式化描述了支持软件重用的操作。给出了反射式软件体系结构的元级和基本级的一致性性质的定义,以及经过重用操作后元级和基本级的一致性的证明方法和过程。

关键词 软件体系结构重用,反射式软件体系结构,一致性

中图法分类号 TP311.5 **文献标识码** A

Study of Consistency for Reflective Software Architecture

LUO Ju-bo[1,2] 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 describes the reuse operatins based on formal specification language—Object-Z. Moreover, it defined the characters of meta-level and base-level of the reflective software architecture. Finally, it provided proof method and process of consistency of base-level and meta-level for the reflective software architecture after the reuse operations.

Keywords Reuse of software architecture, Reflective software architecture, Consistency

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,但是这些重用方法存在以下问题: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[5,6]给出的基于反射机制的软件体系结构重用方法——ArchBean方法将元信息、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。其重用对象是体系结构层的组件、连接器、体系结构片段、风格,因此它是一种更通用、更便捷的体系结构制品本身的重用方法,具有统一的体系结构建模方法的信息。

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;以删除链接操作为例,基于Object-Z形式化描述支持软件重用的操作;为了保证重用后软件体系结构的一致性和正确性,给出反射式软件体系结构的元级和基本级的一致性性质的定义,以及经过重用操作后元级和基本级的一致性的证明方法和过程。

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的元信息,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[6],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。

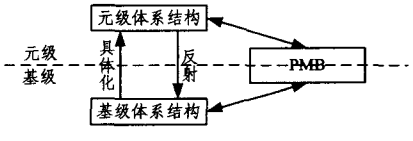

图1 反射式软件体系结构

2.1 基本级体系结构

到稿日期:2009-10-13 返修日期:2009-12-28 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。

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。我们以C2SADEL为例来进行说明，如图2所示。

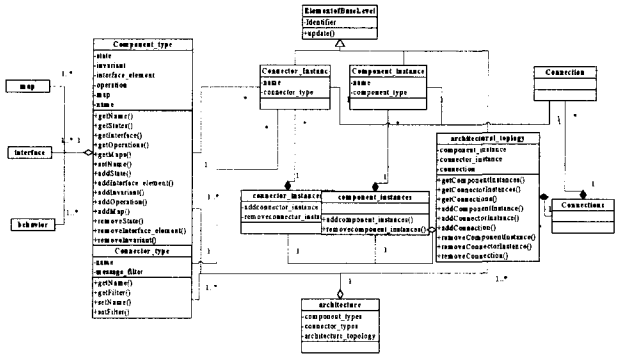

图2　基级体系结构(针对C2SADL)

2.2　元级体系结构

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，元级体系结构如图3所示。

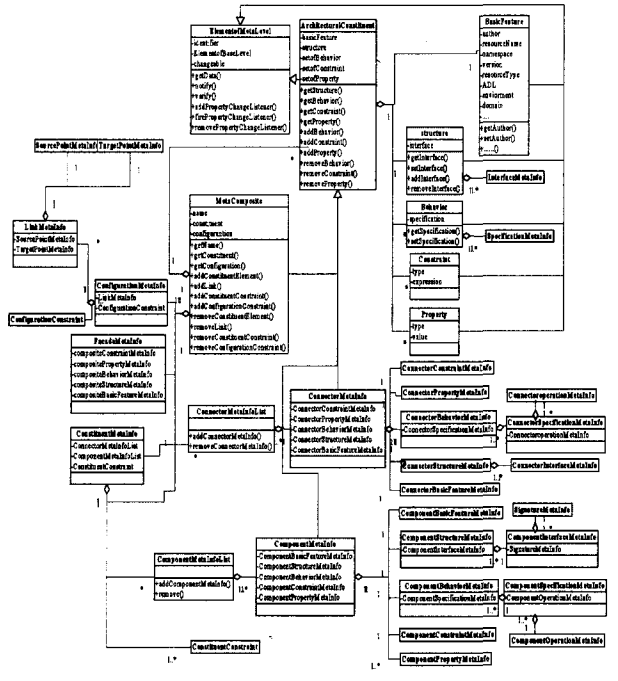

图3　元级体系结构

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，用于完成元级体系结构的更新。

2.3　元级和基级因果关联

元级和基级的因果关联是由PMB协议来实现的，PMB协议包括：定义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，是动态的；定义元级和基级的静态因果关联。

下面给出元级和基级的静态因果关联的形式化描述：

元组件和与其具有因果关联的组件的映射关系的模式*MetacomToInstance*定义：

MetaComToInstance

metacomponent : *MetaComponent*
component _ instance : *Component _ instance*
list _ metacomponent : *List _ MetaComponent*
list _ component _ instance : *List _ Component _ instance*
MetaComToInstance : *MetaComponent* ↦ *Component _ instance*

---

domMetaComToInstance ⊆ *list _ metacomponent*
ranMetaComToInstance ⊆ *list _ component _ instance*
metacomponent.metacomstructure.ComponentName = '*meta*'+ *component _ instance.name*
metacomponent.metacomstructure.TypeName = *component _ instance.component _ type*
metacomponent.metacomstructure.metacominterface
= *metacomponent.metacomstructure.setInterface*(*component _ instance.component _ type.interface _ element*)
metacomponent.metacomstructure.Changeable
= *component _ instance.componentstructure.Changeable*
metacomponent.metacomponentbehavior.Changeable
= *component _ instance.componentbehavior.Changeable*
metacomponent.metacomponentbehavior.metacombehaviorspec
= *metacomponent.metacomponentbehavior.setSpecification*(*component _ instance.component _ type.operation*)
metacomponent.metacomconstraint
= *metacomponent.setConstraint*(*component _ instance.component _ type.invariant*)
metacomponent.metacomproperty
= *metacomponent.setProperty*(*component _ instance.component _ type.state*)

元连接器和与其具有因果关联的连接器的映射关系的模式*MetaconnToInstance*定义：

MetaConnToInstance

metaconnector : *MetaConnector*
connector _ instance : *Connector _ instance*
list _ metaconnector : *List _ MetaConnector*
list _ connector _ instance : *List _ Connector _ instance*
MetaConnToInstance : *MetaConnector* ↦ *Connector _ instance*

---

domMetaComToInstance ⊆ *list _ metaconnector*
ranMetaComToInstance ⊆ *list _ connector _ instance*
metaconnector.message _ filters
= *metaconnector.setFilteringMode*(*connector _ instance.connector _ type.message _ filters*)
metaconnector.metaconnstructure.ConnectorName = '*meta*'+ *connector _ instance.name*
metaconnector.metaconnstructure.TypeName = *connector _ instance.connector _ type*
metaconnector.metaconnstructure.metaconninterface
= *metaconnector.metaconnstructure.setInterface*
(*connector _ instance.connector _ type.interface _ element*)
metaconnector.metaconnstructure.Changeable
= *connector _ instance.metaconnstructure.Changeable*
metaconnector.connectorbehavior.Changeable
= *connector _ instance.connectorbehavior.Changeable*
metaconnector.metaconnbehavior.SpecificationMetaInfo
= *metaconnector.metaconnectorbehavior.setSpecification*
(*connector _ instance.connector _ type.operation*)
metaconnector.metaconnconstraint
= *metaconnector.setConstraint*(*connector _ instance.connector _ type.invariant*)
metaconnector.metaconnproperty
= *metaconnector.setProperty*(*connector _ instance.connector _ type.state*)

3　支持软件重用的操作

支持软件重用的操作包括：增加组件、删除组件、增加连接器、删除连接器、增加链接和删除链接操作。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。

经过任何一个或几个重用操作后，反射式软件体系结构的元级和基级的分量会发生变化，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，下面以删除链接操作为代表，详细给出删除链接重用操作的形式化描述。

删除链接操作 deleteLink，删除体系结构的两个构成元素接口间的链接。我们定义删除组件和连接器之间链接操作的前置条件模式为 Pre_deleteLink_One，操作模式为 deleteLink_One。

前置条件模式 Pre_deleteLink_One 的定义如下：

Pre_deleteLink_One

ElementofMetaLevel
Link
metacomposite : *MetaComposite*
MetaConnector01, *MetaConnector02* : *MetaConnector*

MetaConnector01.verify(*MetaConnector01.metacomconstraint*.*Constraint*.*Expression*) = *true* ∧
MetaConnector02.verify(*MetaConnector02.metaconnconstraint*.*Constraint*.*Expression*) = *true* ∧
metacomposite.verify(*metacomposite.metacompoconstituent.metacompoconsticonstr*.*Constraint*.*Expression*) = *true* ∧
metacomposite.verify(*metacomposite.metacompoconfig.configconstraint*.*Constraint*.*Expression*) = *true*
(*MetaComponent01* ↦ *MetaConnector01*) ∈ *LinkComConn*

操作模式 deleteLink_One 的定义如下：

deleteLink_One

ElemeneofBaseLevel
ElementofMetaLevel
Δ*Link*
Δ*Connections*
architecture : *Architecture*
architectural_topology : *Architectural_topology*
MetaConnector01 : *MetaConnector*; *Connector01* : *Connector_instance*
MetaComponent01 : *MetaComponent*; *Component01* : *Component_instance*
metamomposite : *MetaComposite*

MetaComponent01.verify(*MetaComponent01.metacomconstraint*.*Constraint*.*Expression*) = *true* ∧
MetaConnector01.verify(*MetaConnector01.metaconnconstraint*.*Constraint*.*Expression*) = *true* ∧
metacomposite.verify(*metacomposite.metacompoconstituent.metacompoconsticonstr*.*Constraint*.*Expression*) = *true* ∧
metacomposite.verify(*metacomposite.metacompoconfig.configconstraint*.*Constraint*.*Expression*) = *true*
(*MetaComponent01* ↦ *MetaConnector01*) ∈ *LinkComConn*
list_metacomponent' = *list_metacomponent*
list_metaconnector' = *list_metaconnector*
LinkConnConn' = *LinkConnConn*
metaconns_top' = *metaconns_top*
metaconns_down' = *metaconns_down*
metacomponents' = *metacomponents* \ {*MetaComponent01*}
metaconnector' = *metaconnector* \ {*MetaConnector01*}
LinkComConn' = *LinkComConn* \ {*MetaComponent01* ↦ *MetaConnector01*}
Connectioncomconn' = *Connectioncomconn*
connector_top_instances' = *connector_top_instances*
connector_down_instances' = *connector_down_instances*
connector_instances' = *connector_instances* \ {*Connector01*}
component_instances' = *component_instances* \ {*Component01*}
Connectioncomconn' = *Connectioncomconn* \ {*Component01* ↦ *Connector01*}
list_component_instances' = *list_component_instances*
list_connector_instances' = *list_metaconnector_instances*
MetaConnToIns' = *MetaConnToIns*
MetaComToIns' = *MetaComToIns*

4 反射式软件体系结构的一致性

4.1 反射式软件体系结构一致性定义

下面给出反射式软件体系结构的一致性定义。

反射式软件体系结构 Reflec-tiveArchitecture的基级 ReflectiveArchitecture_BaseLevel 和元级ReflectiveArchitecture_MetaLevel是一致的当且仅当以下 5 个规则全部满足：

规则 1　反射式软件体系结构中不存在孤立的元组件、元连接器、组件和连接器，即从元组件 *MetaComponent* 到与其具有因果关联的组件 *Component_instance* 的映射关系 *MetacomToInstance*：*MetaComponent* ↔ *Component_instance* 是全双射函数，从元连接器 *MetaConnector* 到与其具有因果关联的连接器 *Connector_instance* 的映射关系 *MetaconnToInstance*：*MetaConnector* ↔ *Connector_instance* 是全双射函数。

规则 2　元组件到与其具有因果关联的组件的映射关系 *MetacomToInstance*：*MetaComponent* ⊢→ *Component_instance* 的每一个序偶（*metacomponent*，*component_instance*），*metacomponent* 和 *component_instance* 具有因果关联关系，即：

metacomponent.metacomstructure.ComponentName = 'meta' + *component_instance.name*
metacomponent.metacomstructure.TypeName = *component_instance.component_type*
metacomponent.metacomstructure.metacominterface
= *metacomponent.metacomstructure.setInterface*(*component_instance.component_type*, *interface_element*)
metacomponent.metacomstructure.Changeable
= *component_instance.componentstructure.Changeable*
metacomponent.metacomponentbehavior.Changeable
= *component_instance.componentbehavior.Changeable*
metacomponent.metacomponentbehavior.metacombehaviorspec
= *metacomponent.metacomponentbehavior.setSpecification*(*component_instance.component_type.operation*)
metacomponent.metacomconstraint
= *metacomponent.setConstraint*(*component_instance.component_type.invariant*)
metacomponent.metacomproperty
= *metacomponent.setProperty*(*component_instance.component_type.state*)

规则 3　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* 的每一个序偶（*metaconnector*，*connector_instance*），*metaconnector* 和 *connector_instance* 具有因果关联关系，即

metaconnector.message_filters
= *metaconnector.setFilteringMode*(*connector_instance.connector_type.message_filters*)
metaconnector.metaconnstructure.ConnectorName = 'meta' + *connector_instance.name*
metaconnector.metaconnstructure.TypeName = *connector_instance.connector_type*
metaconnector.metaconnstructure.metaconnintreface
= *metaconnector.metaconnstructure.setInterface*
(*connector_instance.connector_type.interface_element*)
metaconnector.metacomstructure.Changeable
= *connector_instance.metaconnstructure.Changeable*
metaconnector.connectorbehavior.Changeable

$=connector\_instance.connectorbehavior.Changeable$

$metaconnector.metaconnbehavior.SpecificationMetaInfo$

$=metaconnector.metaconnectorbehavior.setSpecification$

$(connector\_instance.connector\_type.operation)$

$metaconnector.metaconnconstraint$

$=metaconnector.setConstraint(connector\_instance.connector\_type.invariant)$

$metaconnector.metaconnproperty$

$=metaconnector.setProperty(connector\_instance.connector\_type.state)$

规则4　元级中元组件到元连接器链接的映射关系 $LinkComConn:MetaComponent \rightarrowtail MetaConnector$ 的每一个序偶($metacomponent$,$metaconnector$),在基级 $ReflectiveArchitecture\_BaseLevel$ 的组件到连接器连接的映射关系 $ConnectionComConn:Component\_instance \rightarrowtail Connector\_instance$ 中都有一个对应的序偶($component\_instance$,$connector\_instance$),并且 $metacomponen$ 和 $component\_instance$ 具有因果关联关系,$metaconnector$ 和 $connector\_instance$ 具有因果关联关系。

规则5　元级中元连接器到元连接器链接的映射关系 $LinkConnConn:MetaConnector \rightarrowtail MetaConnector$ 的每一个序偶($metaconnector01$,$metaconnector02$),在基级连接器到连接器连接的映射关系 $ConnectionConnConn:Connector\_instance \rightarrowtail Connector\_instance$ 中都有一个对应的序偶($connector\_instance01$,$connector\_instance02$),并且 $metaconnector01$ 和 $connector\_instance01$ 具有因果关联关系,$metaconnector02$ 和 $connector\_instance02$ 具有因果关联关系。

4.2　反射式软件体系结构的一致性证明

以重用操作删除链接操作为例来说明反射式软件体系结构的一致性问题。删除链接操作 deleteLink,是删除体系结构的两个构成元素接口间的链接。前面定义了删除组件和连接器之间链接操作的操作模式 deleteLinkt_One,下面先给出删除链接操作后 ReflectiveArchitecture 的基级和元级保持一致性的定理,然后证明之。

定理1　对反射式软件体系结构 ReflectiveArchitecture 进行删除链接操作后,ReflectiveArchitecture 的基级和元级保持一致性。

定理的证明过程如下。

推理删除链接操作模式 addLinkt_One 后,得到反射式软件体系结构元级的各个分量为:

$list\_metacomponent' = list\_metacomponent$

$list\_metaconnector' = list\_metaconnector$

$LinkConnConn' = LinkConnConn$

$metaconns\_top' = metaconns\_top$

$metaconns\_down' = metaconns\_down$

$metacomponents' = metacomponents \backslash \{MetaComponent01\}$

$metaconnector' = metaconnector \backslash \{MetaConnector01\}$

$LinkComConn' = LinkComConn \backslash \{MetaComponent01 \rightarrowtail MetaConnector01\}$　(a)

反射式软件体系结构基级的各个分量为:

$Connectionconnconn' = Connectionconnconn$

$connector\_top\_instances' = connector\_top\_instances$

$connector\_down\_instances' = connector\_down\_instances$

$connector\_instances' = connector\_instances \backslash \{Connector01\}$

$component\_instances' = component\_instances \backslash \{Component01\}$

$Connectioncomconn' = Connectioncomconn \backslash \{Component01 \rightarrowtail Connector01\}$

$list\_component\_instances' = list\_component\_instances$

$list\_connector\_instances' = list\_connector\_instances$　(b)

反射式软件体系结构的元组件到组件的映射关系和元连接器到连接器的映射关系的变化如下:

$MetaConnToInstance' = MetaConnToInstance$

$MetaComToInstance' = MetaComToInstance$　(c)

分析式(a)和式(b),按形式化方法推导出元级删除了一个元组件与元连接器的链接 $MetaComponent01 \rightarrowtail MetaConnector01$,$metacomponents$ 删除了一个元组件 $MetaComponent01$,$metaconnectors$ 删除了一个元连接器 $MetaConnector01$,其他分量没有变化;基级删除了一个组件和连接器的连接 $Component01 \rightarrowtail Connector01$,$component\_instances$ 删除了一个组件 $Component01$,$connector\_instances$ 删除了一个连接器 $Connector01$,其他分量没有变化。

$MetaComponent01$ 和 $Component01$ 因为具有因果关联关系,并且有式(c)中的 $MetaComToInstance' = MetaComToInstance$,所以满足规则2的定义。

又 $MetaConnector01$ 和 $Connector01$ 因为具有因果关联关系,并且有式(c)中的 $MetaConnToInstance' = MetaConnToInstance$,所以满足规则3的定义。

$MetaComponent01$ 和 $Component01$ 具有因果关联关系,$MetaConnector01$ 和 $Connector01$ 具有因果关联关系,故 $MetaComponent01 \rightarrowtail MetaConnector01$ 和 $Component01 \rightarrowtail Connector01$ 具有因果关联关系,所以满足规则4的定义。

元连接器到元连接器连接的映射关系 $LinkConnConn:MetaConnector \rightarrowtail MetaConnector$ 没有变化,因此满足规则5的定义。

由于元组件、元连接器、组件和连接器都没有增加和删除,故反射式软件体系结构中不存在孤立的元组件、元连接器、组件和连接器,所以满足规则1的定义。

综合以上分析和证明,反射式软件体系结构满足其一致性定义的所有规则,所以可以下结论:删除链接操作 addLinkt_One作用于反射式软件体系结构后,反射式软件体系结构的基级和元级是一致的。

同理可以证明删除两个连接器之间的链接后,ReflectiveArchitecture 的基级和元级保持一致性。

所以得出结论:对反射式软件体系结构 ReflectiveArchitecture 进行删除链接操作后,ReflectiveArchitecture 的基级和元级保持一致性。

同样可以证明增加组件、删除组件、增加连接器、删除连接器和增加链接操作作用于反射式软件体系结构后,反射式软件体系结构的基级和元级是一致的。

5　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

(下转第233页)

Deep Web 信息集成和模式匹配过程中存在的不确定知识和推理问题。例如，可以采用 DFDLs 扩展的 OWL 来描述股票市场某只股票的变化趋势情况。如下所示：

```
〈owl:Class rdf:ID="Stock"〉
    〈owl:unionOf rdf:parse Type="Collection"〉
        〈owl:Class rdf:about="#000001"
    Type="→" fuzzy:degree="0.8"/〉
    〈owl :Class rdf:about="#000002"
            Type="←" fuzzy:degree="0.75"/〉
        〈/owl:unionof〉
    〈/owl:Class〉
```

结束语 本文在 Deep Web 中的不确定知识的动态性、模糊性等方面的基础上，研究了将传统描述逻辑和动态模糊理论相融合，建立一套基于动态模糊逻辑的面向 Deep Web 的不确定知识的表示、模型生成的理论与方法，即动态模糊描述逻辑（DFDLs），并提出了一种基于动态模糊 tableau 的 DFDLs 的 ABox 约束下的可满足性推理算法。DFDLs 可以用于面向 Deep Web 的不确定知识的合理推理和利用，它弥补了描述逻辑在模糊性和动态性方面的不足，既为语义 Web 提供了进一步的逻辑支撑，也为表示和利用 Deep Web 中动态和模糊等不确定知识提供了强有力的理论基础。提出有效的推理算法并分析推理算法的可判断性和时间复杂性，是今后需要进一步研究的问题。

参 考 文 献

[1] Barbosa L, Freire J. Siphoning Hidden-Web Data through Keyword-based Interfaces[C]// Brazilian Symposium on Databases (SBBD). 2004

[2] Ghanem T M, Aref W G. Databases Deepen the Web[J]. IEEE Computer, 2004, 73(1): 116-117

[3] He B K, Chang C-C, Han J. Mining Complex Matchings Across Web Query Interfaces[C]// Proceedings of the 9th ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery(SIGMOD-DMKD'04). Paris, France, June 2004

[4] Baader F, Nutt W. Basic Description Logic [M]. Handbook of Description Logic. Cambridge University Press, January 2003

[5] Horrocks I. Using an expressive description logic: FaCT or fiction? [C]// Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning(KR'98). 2001: 199-204

[6] Horrocks I, Patel-Schneider P F. Reducing OWL entailment to description logic satisfiability[C]// the 2003 International Semantic Web Conference(ISWC 2003). 2003: 17-29

[7] Hendler J, McGuinness D L. The DARPA Agent Markup Language[J]. IEEE Intelligent Systems, 2000, 15(6): 67-73

[8] Chang Liang, Shi Zhongzhi, Qiu Lirong, et al. Dynamic Description Logic: Embracing Actions into Description Logic[C]// Proceedings of the 2007 International Workshop on Description Logics. 2007

[9] 李凡长，刘贵全，佘玉梅. 动态模糊逻辑引论[M]. 昆明：云南科技出版社，2005

[10] 史忠植，董明楷，蒋运承，等. 语义 Web 的逻辑基础[J]. 中国科学(E辑)，2004，34(10)：1123-1138

[11] 蒋运承，史忠植，汤庸，等. 面向语义 Web 语义表示的模糊描述逻辑[J]. 软件学报，2007，18(6)：1257-1269

[12] 梅婧，林作铨. 从 ALC 到 SHOQ(D)：描述逻辑及其 Tableau 算法[J]. 计算机科学，2005，32(3)：1-11

[13] 王驹，蒋运承，唐素勤. 一种模糊动态描述逻辑[J]. 计算机科学与探索，2007，1(2)：216-227

(上接第 160 页)

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。

我们提出的基于反射机制的软件体系结构重用方法——ArchBean 方法是一种更通用、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，可以完成设计阶段软件体系结构制品的重用；为了保证重用软件体系结构的正确性和一致性，本文给出反射式软件体系结构的元级和基本级的一致性性质的定义，以及经过重用操作后，元级和基本级的一致性的证明方法和过程。所以本文以及后续的研究内容在一定程度上可以较好地解决软件体系结构重用存在的主要问题。

结束语 本文的研究工作与已有研究的不同之处在于：(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)基于 Object-Z 形式化描述支持软件重用的部分操作；给出反射式软件体系结构的元级和基本级的一致性性质的定义，以及经过重用操作后元级和基本级的一致性的证明方法和过程。

作为今后的工作，我们将完善 PMB 协议的定义，开发一个基于 Eclipse 插件技术的支撑工具 ArchBean Studio 来实现反射式软件体系结构的基于 Object-Z 形式化描述，以及自动实现基级和元级的一致性验证。

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 and Kerth, eds. Addison-Wesley, 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. Sorenson. Designing object-oriented frameworks, In CRC Handbook of Object Technology [M]. CRC Press, 1998

[5] 叶鹏，应时，罗巨波. 一种支持软件体系结构重用的元信息模型[J]. 计算机科学，2009，36(5)：145-150

[6] 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