ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.8, August 2009, pp.2113−2123
doi: 10.3724/SP.J.1001.2009.00573

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

以体系结构为中心的模型转换的语义描述框架*

侯金奎[1,2+], 王海洋[1], 马 军[1], 万建成[1], 杨 潇[1]

[1](山东大学 计算机科学与技术学院,山东 济南 250101)

[2](潍坊学院 计算机与通信工程学院,山东 潍坊 261061)

Semantic Description Framework for Architecture-Centric Model Transformation

HOU Jin-Kui[1,2+], WANG Hai-Yang[1], MA Jun[1], WAN Jian-Cheng[1], YANG Xiao[1]

[1](School of Computer Science and Technology, Shandong University, Ji'nan 250101, China)

[2](School of Computer and Communication Engineering, Weifang University, Weifang 261061, China)

+ Corresponding author: E-mail: hjk@mail.sdu.edu.cn

Hou JK, Wang HY, Ma J, Wan JC, Yang X. Semantic description framework for architecture-centric model transformation. *Journal of Software*, 2009,20(8):2113−2123. http://www.jos.org.cn/1000-9825/573.htm

Abstract: In this paper, the typed category theory is extended and combined with process algebra to provide a unified description framework for the formal semantics of architecture-centric model transformations. The structural semantics of architecture models are described within typed category diagrams, and the behavioral semantics are represented by process traces affiliated to the categorical framework, and the mapping relations between component models are formally described by morphisms and functors of category theory. The framework can be used for the description, analysis and judgment of property preservation of model transformations, and thus make an effective support for model-driven software development.

Key words: model-driven development; model transformation; software architecture; semantic description

摘 要: 在对类型范畴理论进行扩展的基础上,将其与进程代数相结合,为软件体系结构模型及其间的转换关系提供了一种统一的语义描述框架.模型的结构语义由类型范畴图表来指代,其行为语义则由范畴附带的进程行为迹来表示,模型间的映射关系用范畴理论中的态射和函子来形式化描述.该描述框架可用于模型转换中特性保持问题的描述、分析和判定,从而为模型驱动的软件开发提供有力的支持.

关键词: 模型驱动开发;模型转换;软件体系结构;语义描述

中图法分类号: TP301 **文献标识码**: A

1 Introduction

Model-Driven development (MDD) has become an active research area of software engineering[1], which deals with the complexity of software development by raising the level of abstraction. The correctness of model

* Supported by the National Natural Science Foundation of China under Grant No.60673130 (国家自然科学基金); the Key Science-Technology Development Project of Shandong Province of China under Grant No.2008GG10001026 (山东省科技攻关项目)

Received 2008-04-22; Accepted 2008-10-07

transformations is a key issue of model-driven engineering. The general criteria about the correctness of model transformations comprise syntactic correctness, syntactic completeness, termination, confluence and semantic consistency[2]. There have already been comparatively mature solutions for the judgment of these criteria with the exception of semantic consistency. How to ensure semantic consistency between the models before and after transformation has become a key issue on the road of MDD becoming more mature.

As the standard for object-oriented modeling, the UML is still largely undefined from a semantic point of view, which brings difficulties in such a scenario due to the ambiguity of models[3]. That makes the modeling concepts and their semantic specifications do not very well suit the starting point for some works of MDD, such as automatic code generation and formal verification. Many researchers[1,3] believe that the current descriptions of high-level models of MDD are neither complete nor accurate for lacking understandable formal semantic meanings, which makes it difficult to achieve automatic model transformations, and also hard to build effective mechanisms for the evaluation and verification on the transformations. The definition, description, and proof of semantic property preservation of model transformation are still problems unresolved[4]. An integrated semantic model for non-formal modeling languages is still missing. The existing describing mechanisms for the constraints of property preservation are all built for some specific scenarios[4−7], which makes them not generic enough for more situations. All the facts show that, the lack of description and calculation approaches for semantic properties currently is the main lacking theory of model-driven software development, and to build a theory for semantic description and calculation becomes the basis and urgency for its healthy and rapid development.

Category theory[8] provides the right level of mathematical abstraction to address languages for describing software architectures, and its abstract framework provide correct semantics for the configuration of complex systems from their component parts[9]. In this paper, based on the work by Fiadeiro and Lopes[9], a unified semantic description framework for architecture-centric models and their transformational relationships is proposed by combining category theory with process algebra. It can be used for the description, analysis and judgment of property preservation of model transformations.

The rest of this paper is organized as follows. In Section 2, formal semantics of component-based architecture models are presented based on category theory and process algebra. Formal description of architecture-centric model mapping is given in Section 3. A case study about a supply chain management system is shown in Section 4 to further explaining the ideas. The paper ends with conclusions and future works.

2 Formal Semantics of Architecture Models

In this paper, category theory is used as a basic framework for the semantic description of architecture models. The basic knowledge about category theory can be found in Ref.[8], which is not repeated here. From the point of view of describing and verifying semantic properties, the distinction between component and connector is often subtle[10]. In order to maintain regularity and simplicity, we do not distinguish between these categories at the specification level, and both component and connector are generically called component.

Figure 1 depicts the architecture of a supply chain management system, which will be used to illustrate the relevant concepts and models throughout the paper. Herein, we simplify the services and omit some details, and only six components are contained: the shopping service (*Shop*), the store service (*Store*), the banking service (*Bank*), the transporting service (*Transport*), the supplying service (*Supplier*) and the Email service (*Email*). The *Shop* and the *Store* are combined together to form a composite component (*ServiceSystem*). The client component (*Client*) can require purchasing goods by calling the method *SellItem* provided by the *Shop*, and it also can require returning goods by calling the method *Recede*. In an ongoing buying, the *Shop* first checks the *store* to make sure

whether the stock is sufficient or not. The buying will fail if the stock is insufficient. If the *Store* keeps the required goods sufficiently, it will subtract the purchase quantity from the total stock. Then, the *Shop* will call the method *ProcPay* provided by the *Bank* to require the payment of the customer. If the stock is insufficient or the total quantity of the required goods is less than a fixed number, the *Store* will place an order to the *Supplier* after checking. Moreover, the *Shop* also can inform the *Bank* of returning the payment of the customer through the method *Compensate*. After the *Bank* confirms the paying, the *Shop* will inform the *Transport* through the method *ShipItem* of transporting the goods to the customer. The *Shop* can also inform the *Transport* through the method *WithDraw* of back transporting the goods. If the buying failed, the *Shop* will inform the Email service of sending an apologetic message to the client component.

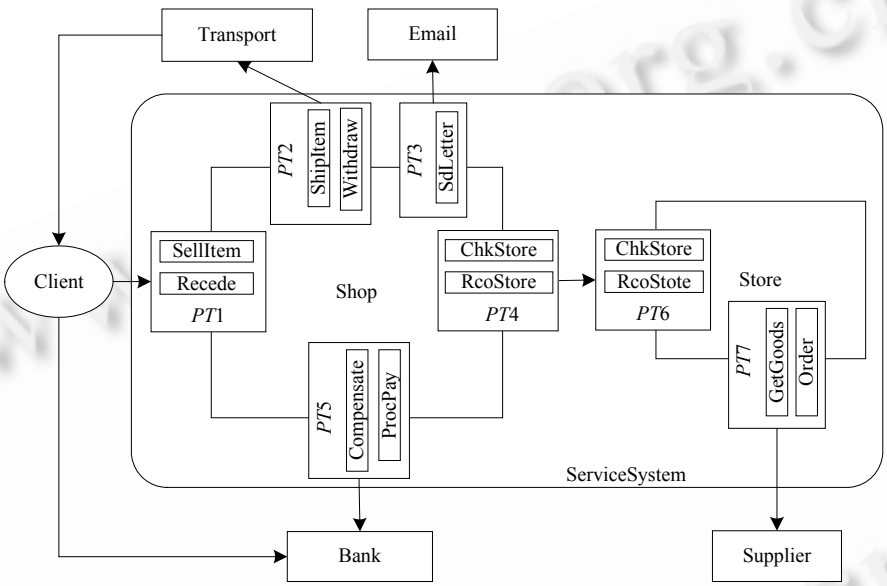

Fig.1 Architecture depiction for a supply chain management system

2.1 Component specification

Process algebra (PA)[10] is used to formally describe the external behaviors of components in this paper. The basic knowledge about PA can be found in Ref.[11]. The external behavioral trace of a component is expressed as a state transition sequence $BP=\langle SS,AS,TS\rangle$, where $SS=\{s_i|0\le i\le n\}$ is a finite set of component states, and s_{Init} and s_{Fina} respectively represent the initial state and the final state of the component, and $AS=\{a_i|0\le i\le m\}$ is a finite set of component actions, and $TS\subseteq SS\times AS\times SS$ is the finite set of state transitions which represent the interactive behavior of the component. The trace $\langle a_1,a_2,\ldots,a_n\rangle$ will be called a complete external behavioral trace iff the transition set TS defined as $s_0\rightarrow_{a1}s_1\rightarrow_{a2}s_2\rightarrow\ldots\rightarrow_{an}s_n$ ($s_0=s_{Init},s_n=s_{Fina}$).

Definition 1 (**component signature**). A component signature is a 8-tuple $\theta=\langle Cid,\Sigma,A,\Gamma,fa,fp,D,BP\rangle$, where

(1) Cid is the unique identifier of the component;

(2) $\Sigma=\langle S,\Omega\rangle$ is a data signature in the usual algebraic sense, i.e., S is a set of sort symbols and Ω is an $S^*\times S$-indexed family of function symbols;

(3) A is an $S^*\times S$-indexed family of attribute symbols, in which each attribute is typed by a data sort in S;

(4) Γ is an S^*-indexed family of port symbols;

(5) fa: $A\rightarrow 2^S$ is a set of total functions, which shows the properties of the attributes;

(6) fp: $\Gamma\rightarrow 2^S$ is a set of total functions, which shows the properties of the ports;

(7) D: $\Gamma \to 2^A$ is a total function, and for each $p \in \Gamma$, $D(p)$ is the collection of attributes that can be affected via the port p;

(8) BP is the description of the external behaviors of the component, which is formally defined using PA.

Definition 2 (**component specification**). A component specification is a pair (θ,Δ), in which θ is a component signature $\langle Cid,\Sigma,A,\Gamma,fa,fp,D,BP\rangle$ and Δ, the body of the specification, is a quadruple (I,F,B,Φ), where

(1) I is a set of Σ-propositions constraining the initial values of the attributes;

(2) F assigns to every port $p \in \Gamma$ a non-deterministic command, which relates all attributes in $D(p)$ to the actions of $G(p)$. Here $G(p)$ represents the set of actions of a port p;

```
Component Shop
  Attributes
      Private Cid: String;
      Private OrderedItem: ItemType;
      Private ChkAvail: Boolean;
      Private PayInfo: Boolean;
      Private ShipInfo: ItemType;
      …
  Ports
      In PT1{
          Boolean SellItem(item: ItemType);
          Boolean Recede(item: ItemType);
      }
      In/Out PT2{
          Boolean ShipItem(item: ItemType);
          Boolean Withdraw(item: ItemType);
      }
      Out PT3{
          Void SdLetter(letter: LetterType);
      }
      In/Out PT4{
          Boolean ChkStore(item: ItemType);
          Boolean RcoStore(item: ItemType);
      }
      In/Out PT5{
          Boolean ProcPay(fee: MoneyType);
          Boolean Compensate(fee: MoneyType);
      }
      …
  Axioms
      SellItem(item)⇒OrderedItem=item;
      ChkAvail=ChkStore(item);
      Withdraw(item)⇒ShipInfo=item;
      PayInfo=ProcPay(fee);
      ShipItem(item)⇒ProcPay(fee)=OK;
      SdLetter(letter)⇒SellItem(item)=FAIL;
      ProcPay(fee)=FAIL⇒RcoStore(item);
      ((ChkStore(item)=FAIL)∧(ProcPay(fee)=OK))=FALSE;
      …
  Behavior
  BPShop Δ= BuyRequest.ChkStore.(ChkOK.RequirePay.(PayFeeOK.
            ShipItem.BPShop+PayFeeFail.RcoStore.SendLetter.
            BPShop)+ChkFail.SendLetter.BPShop)+RecedeRequest.
            Withdraw.Compensate.RcoStore.BPShop;
  …
```

Fig.2 Specification of the component *Shop*

(3) B assigns to every port $p \in \Gamma$ a Σ-proposition as its guard, which represents the conditions and constraints that should be satisfied for achieving the objectives of the component;

(4) Φ is a finite set of θ-formulae (the axioms of the description), which represents the functional and non-functional objectives of the component.

The component specification for the *Shop* in Fig.1, denoted as CP_{Shop}, is shown in Fig.2. In the specification $CP_{Shop}=(\theta_{Shop},\Delta_{Shop})$, $\Delta_{Shop}=(I_{Shop},F_{Shop},B_{Shop},\Phi_{Shop})$ is shown by the Axioms part. Herein, the axiom "$ShipItem \Rightarrow ProcPay(fee)$=OK" indicates that the goods can be transported only after the customer has paid successfully. The component signature $\theta_{Shop}=\langle Cid_{Shop},\Sigma_{Shop},A_{Shop},\Gamma_{Shop},fa_{Shop},fp_{Shop},D_{Shop},BP_{Shop}\rangle$, where Σ_{Shop} is the data signature; $A_{Shop}=\{OrderedItem, ChkAvail, PayInfo, Shipinfo\}$; $\Gamma_{Shop}=\{PT1, PT2, PT3, PT4, PT5\}$; fa_{Shop} describes the information of the attribute types, etc. Herein, $type(OrderedItem)=ItemType$, $type(ChkAvail)=Boolean$, $type(PayInfo)=Boolean$, $type(ShipInfo)=ItemType$; fp_{Shop} describes the information of port types, the types of received messages, etc. Herein, $type(PT1)=In$, $MessageType(PT1)=ItemType$, $type(PT2)=In/Out$, $MessageType(PT2)=ItemType$. The description of D_{Shop} contains: $D_{Shop}(PT1)=\{OrderedItem\}$, $D_{Shop}(PT2)=\{Shipinfo\}$, $D_{Shop}(PT3)=\{\ \}$, $D_{Shop}(PT4)=\{ChkAvail\}$, $D_{Shop}(PT5)=\{PayInfo\}$. BP_{Shop} is the behavioral description, in which *BuyRequest*, *RecedeRequest*, *ChkStore*, … are the actions of the component.

2.2 Component specification morphism

The relationships between component specifications are represented by morphisms of category theory.

Definition 3 (**component signature morphism**). Given two component signatures $\theta_1=\langle Cid_1,\Sigma_1,A_1,\Gamma_1,fa_1,fp_1,D_1,BP_1\rangle$ and $\theta_2=\langle Cid_2,\Sigma_2,A_2,\Gamma_2,fa_2,fp_2,D_2,BP_2\rangle$, a morphism from θ_1 to θ_2, denoted by σ: $\theta_1 \to \theta_2$, consists of

(1) An algebraic signature mapping σ_{Σ}: $\Sigma_1 \to \Sigma_2$;

(2) An attribute mapping σ_{at}: $A_1 \to A_2$, such that,

(2.1) For some attributes f: $s_1,\ldots,s_n \to s$ in A_1, there exist attribute symbols $\sigma_{at}(f)$: $\sigma_{\Sigma}(s_1),\ldots, \sigma_{\Sigma}(s_n) \to \sigma_{\Sigma}(s)$ in A_2;

(2.2) $\exists a \in A_1, fa_1(a) =_D fa_2(\sigma_{at}(a))$, where $=_D$ means the consistency relations between property descriptions;

(3) A port mapping σ_{ac}: $\Gamma_1 \to \Gamma_2$, such that,

(3.1) for some ports p: $s_1,\ldots,s_n$ in Γ_1, there exist port symbols $\sigma_{ac}(p)$: $\sigma_{\Sigma}(s_1), \ldots, \sigma_{\Sigma}(s_n)$ in Γ_2;

(3.2) $\exists p \in \Gamma_1, fp_1(p) =_D fp_2(\sigma_{ac}(p))$;

(4) $\exists p \in \Gamma_1, \sigma_{at}(D_1(p)) \subseteq D_2(\sigma_{ac}(p))$;

(5) A behavioral description mapping σ_{BP}: $BP_1 \to BP_2$.

Conditions (1)~(3) show that the component signature morphism consists of a port mapping and an attribute mapping, and the consistency between their property descriptions should be preserved. The fourth condition guarantees that the attributes affected by a certain port must be preserved through the morphism. The last condition shows the mapping relations between the behavioral descriptions of the two components.

Definition 4 (**component specification morphism**). Given two component specifications $CP_1=\langle \theta_1,\Delta_1\rangle$ and $CP_2=\langle \theta_2,\Delta_2\rangle$, where $\theta_1=\langle Cid_1,\Sigma_1,A_1,\Gamma_1,fa_1,fp_1,D_1,BP_1\rangle$, $\Delta_1=(I_1,F_1,B_1,\Phi_1)$, $\theta_2=\langle Cid_2,\Sigma_2,A_2,\Gamma_2,fa_2,fp_2,D_2,BP_2\rangle$, $\Delta_2=(I_2,F_2,B_2,\Phi_2)$, a morphism from CP_1 to CP_2, denoted by ω: $CP_1 \to CP_2$, is a signature morphism σ: $\theta_1 \to \theta_2$, such that

(1) $\exists q \in \Phi_1, \omega(q) \in \Phi_2$;

(2) $\exists p_1 \in \Gamma_1, a_1 \in D_1(p_1), F_2(\sigma(p_1), \sigma(a_1)) \Rightarrow \omega(F_1(p_1,a_1))$;

(3) $\exists q \in I_1, \omega(q) \in I_2$;

(4) $\exists p_1 \in \Gamma_1, B_2(\sigma(p_1)) \supseteq \omega(B_1(p_1))$.

The first condition given above guarantees the functional and non-functional objectives should be preserved. The second condition means that the effects of the relevant instructions can only be preserved or made more deterministic, and the third condition indicates that some initialization conditions are preserved. The last condition allows relevant guards to be strengthened but not to be weakened.

2.3 Hierarchical composition of component model

A composite component is constructed from interconnecting instances of more primitive components, which defines a configuration. Category theory supports this kind of hierarchical design, in which the colimit can be used for defining composition operations between components. In this way, a composite component is described as a categorical diagram involving component specifications and specification morphisms. The objects are components and the arrows (morphisms) indicate how the components are interconnected. The morphism types imply the different semantics of component relations. Interaction morphism determines how a component's service is combined with the services provided by other components in the system. Composition morphism depicts a kind of structure-preserving mappings, which is used to establish the relationship that must exist between two component descriptions so that one of them may be considered as a sub-component of the other.

The behavioral semantics of a composite component can be computed based on its configuration and the behaviors of its subcomponents, in which the parallel composition operator "$\_\|_{\pi}\_$" of PA[10] can be used. For example, given two connected components $\theta_i=\langle Cid_i,\Sigma_i,A_i,\Gamma_i,fa_i,fp_i,D_i,BP_i\rangle$ and $\theta_j=\langle Cid_j,\Sigma_j,A_j,\Gamma_j,fa_j,fp_j,D_j,BP_j\rangle$, we can use $BP_i \|_{\pi(\theta_i,\theta_j)} BP_j$ to represent the composite behavior iff the collection of interaction ports between the two components are expressed by $\pi(\theta_i,\theta_j)$. A group of interacting components $\{\theta_1,\theta_2,\ldots,\theta_n\}$ ($\theta_i=\langle Cid_i,\Sigma_i,A_i,\Gamma_i,fa_i,fp_i,D_i,BP_i\rangle$) can be assembled into an architecture, and the behavioral semantics of the whole architecture can be

formalized as $BP_1 \|_{\pi(\theta_1,\theta_2)} BP_2 \|_{\pi(\theta_1,\theta_3)\cup\pi(\theta_2,\theta_3)} BP_3 \| \ldots \|_{\pi(\theta_1,\theta_n)\cup\ldots\cup\pi(\theta_{n-1},\theta_n)} BP_n$.

Definition 5 (**colimit of component signatures**). Given two component signatures $\theta_1=\langle Cid_1,\Sigma_1,A_1,\Gamma_1,fa_1,fp_1,D_1,BP_1\rangle$ and $\theta_2=\langle Cid_2,\Sigma_2,A_2,\Gamma_2,fa_2,fp_2,D_2,BP_2\rangle$, the colimit of θ_1 and θ_2 is given by the signature $\theta=\theta_1\|\theta_2=\langle Cid,\Sigma,A,\Gamma,fa,fp,D,BP\rangle$ and two composition morphisms σ_1: $\theta_1\to\theta$ and σ_2: $\theta_2\to\theta$, where

(1) $(\Sigma, \sigma_{\Sigma_1}, \sigma_{\Sigma_2})$ is the amalgamated sum of Σ_1 and Σ_2, in which $\sigma_{\Sigma_1}: \Sigma_1\to\Sigma$ and $\sigma_{\Sigma_2}: \Sigma_2\to\Sigma$;

(2) $(A, \sigma_{ac_1}, \sigma_{ac_2})$ is the amalgamated sum of A_1 and A_2, in which $\sigma_{ac_1}: A_1\to A$ and $\sigma_{ac_2}: A_2\to A$;

(3) $(\Gamma, \sigma_{at_1}, \sigma_{at_2})$ is the amalgamated sum of Γ_1 and Γ_2, in which $\sigma_{at_1}: \Gamma_1\to\Gamma$ and $\sigma_{at_2}: \Gamma_2\to\Gamma$;

(4) $\forall a_i\in A_i$, i=1,2, $fa(\sigma_{at_i}(a_i))=fa_i(a_i)$;

(5) $\forall p_i\in\Gamma_i$, i=1,2, $fp(\sigma_{ac_i}(p_i))=fp_i(p_i)$;

(6) $\forall p_i\in\Gamma_i$, i=1,2, $D(\sigma_i(p_i))=\sigma_i(D_i(p_i))$;

(7) $BP=BP_1\|_{\pi(\theta_1,\theta_2)} BP_2/\{e|e\in\pi(\theta_1,\theta_2)\}$, where "_/_" is the hiding operator of PA[10].

Definition 6 (**colimit of component specifications**). Given two component specifications $CP_1=\langle\theta_1,\Delta_1\rangle$ and $CP_2=\langle\theta_2,\Delta_2\rangle$, in which $\theta_1=\langle Cid_1,\Sigma_1,A_1,\Gamma_1,fa_1,fp_1,D_1,BP_1\rangle$, $\Delta_1=(I_1,F_1,B_1,\Phi_1)$, $\theta_2=\langle Cid_2,\Sigma_2,A_2,\Gamma_2,fa_2,fp_2,D_2,BP_2\rangle$, $\Delta_2=(I_2,F_2,B_2,\Phi_2)$, the colimit of CP_1 and CP_2 is given by the specifications $CP=CP_1\|CP_2=(\theta,\Delta)$ and two composition morphisms ω_1: $CP_1\to CP$ and ω_2: $CP_2\to CP$, where

(1) $\theta=\theta_1\|\theta_2$, σ_1: $\theta_1\to\theta$ and σ_2: $\theta_2\to\theta$ constitute the colimit of θ_1 and θ_2;

(2) $\Delta=(I,F,B,\Phi)$ is computed as follows:

(2.1) $I=\omega_1(I_1)\cup\omega_2(I_2)$;

(2.2) $\forall p_i\in\Gamma_i$, $a_i\in D_i(p_i)$, i=1,2, $F(\sigma_i(p_i), \sigma_i(a_i))=\omega_i(F_i(p_i,a_i))$;

(2.3) $\forall p_i\in\Gamma_i$, i=1,2, $B(\sigma_i(p_i))=\omega_i(B_i(p_i))$;

(2.4) $\Phi=\Phi_1\cup\Phi_2$.

2.4 Architecture model

Typed category proposed by Lu[11] adds some representational and inferential power to the category theory, but does not break the basic framework of category theory. In this paper, the typed category is extended further by adding types to both the objects and the morphisms, and each type can be defined with a series of features. Thereby, a bijective mapping from the concepts of software architecture to the ones of the typed category can be defined as follows: component instance to object, component relation to morphism, component specification to object type, relation between component specifications to morphism type, component properties to features of object types, properties of component relations to features of morphism types. In this way, a software architecture model can be expressed as a typed category.

Definition 7 (**architecture model**). An architecture model is a 5-tuple $AM=\langle CO,CR,CT,RT,RuleS\rangle$, where CO is a collection of component instances as objects; CR is a collection of component-relationship instances as object morphisms defined over CO; CT is a collection of component specifications; RT is a collection of specification morphisms as relation-types defined over CT; $RuleS$ is the set of rules for relation-type composition. In addition to the basic conditions of the definition of category[8], the following conditions also have to be satisfied:

(1) $CO=\{o_i|1\le i\le n, sort(o_i)\in CT\}$, where *sort* represents a function which returns the type of an object;

(2) $CR=\{r_j|0\le j\le m, \exists a,b\in CO, r_j=(a,b,t), t=sort(r_j)\in RT\}$;

(3) $RuleS$: $RT\times RT\to RT$, and for all $(t,s)\in dom(RuleS)$, the type $w=t\times s$ is called the composed type of t and s; and for all $r_i,r_j\in CR$, $r_i=(a,b,t)$, $r_j=(b,c,s)$, there exists a composed morphism $r_k=(a,c,w)=r_j\circ r_i\in CR$;

(4) For each $a\in CO$, there exists an identity morphism $r_a=(a,a,u)$, $u=sort(r_a)\in RT$; and for all $t\in RT$, $u\times t=t\times u=t$ hold;

(5) For all $r_i=(a,b,t)$, $r_j=(b,c,s)$, $r_k=(c,d,q)$, $r_k\circ(r_j\circ r_i)=(a,d,(t\times s)\times q)=(a,d,t\times(s\times q))=(r_k\circ r_j)\circ r_i$;

(6) For all $r_i=(a,b,t)\in CR$, $r_i\circ r_a=r_b\circ r_i=r_i$, where $r_a=(a,a,u)$, $r_b=(b,b,u)$.

In our notation of categorical diagram, an architecture model is a typed category composed of component specifications and their morphisms. The specification of the whole system configurations is given by the colimit of the underlying diagrams. The semantics of the configuration diagram should be seen as an abstraction of the cooperative execution that is obtained by coordinating the local executions according to the interconnections.

3 Architecture Model Mapping and Semantic Property Preservation

In this paper, model mapping especially represents the mapping relations from the component specifications at a higher abstract level to the specifications at a lower one, which also can be formally described by morphisms of category theory (called mapping morphisms). Category theory also provides us with the means to establish the relationships between architectural models at different abstract levels: functors.

Definition 8 (**architecture mapping functor**). An architecture mapping functor from the architecture model $AM_1=\langle CO_1,CR_1,CT_1,RT_1,RuleS_1\rangle$ to $AM_2=\langle CO_2,CR_2,CT_2,RT_2,RuleS_2\rangle$, denoted by Fu: $AM_1\rightarrow AM_2$, is a function that satisfies the following:

(1) For every component object $cs\in CO_1$, $Fu(cs)\in CO_2$;

(2) For every component specification $cp\in CT_1$, $Fu(cp)\in CT_2$;

(3) Fu is a homomorphism from RT_1 to RT_2 with the following properties:

(3.1) Fu associates each type $t\in RT_1$ with a type $Fu(t)\in RT_2$;

(3.2) For each unit type of identity morphisms $u\in RT_1$, $u=Fu(u)\in RT_2$;

(3.3) For all $t,s\in RT_1$, always $Fu(t),Fu(s)\in RT_2$, and $Fu(t\times s)=Fu(t)\times Fu(s)$;

(4) Fu is a homomorphism from CR_1 to CR_2 with the following properties:

(4.1) For all $cs,cs'\in CO_1$, $(cs,cs',t)\in CR_1$ implies that $(Fu(cs),Fu(cs'),Fu(t))\in CR_2$;

(4.2) For all $r_a=(a,a,u)\in CR_1$, $u=sort(r_a)\in RT_1$, always $Fu(r_a)=(Fu(a),Fu(a),Fu(u))=(Fu(a),Fu(a),u)\in CR_2$;

(4.3) $Fu(f\circ g)=Fu(f)\circ Fu(g)$, whenever $g,f\in CR_1$ and $f\circ g$ is defined.

Model-driven development can be regarded as a multi-level architecture space composed of architecture models at different levels of abstraction, and the development process can be considered as a series of architecture-centric model transformations that preserve the design decisions and semantic properties of source models.

The formal description of architecture-centric model mappings can be used to judge whether a transformation satisfies some property preservation constraints or not. In typed category based architecture model, the structural semantics is represented within categorical diagrams which depicts the architecture configuration and specifies the components and their relations. In order to analyze the impact of a model transformation on the organizational structure of the system, we can first analyze the impact on dependency relations of components according to their interconnections. Let CO_S be the set of components defined in the source architecture model AM_S, and $CR_S=\{\langle c_m,c_n\rangle|c_m,c_n\in CO_S,\ m\neq n\}$ be the set of component relations, and $G_S=\langle V_S,E_S\rangle$ be the corresponding categorical diagrams where $V_S\leftrightarrow CO_S$, $E_S\leftrightarrow CR_S$. Similarly, the categorical diagrams for the target architecture model $AM_T=\langle CO_T,CR_T\rangle$ is represented as $G_T=\langle V_T,E_T\rangle$, where $V_T\leftrightarrow CO_T$, $E_T\leftrightarrow CR_T$. The following is the algorithm to judge whether the dependency relations are consistent or not.

Algorithm 1. To judge whether the dependency relations of components are consistent between before and after model transformation.

Inputs: two categorical diagrams $G_S=\langle V_S,E_S\rangle$ and $G_T=\langle V_T,E_T\rangle$, which respectively represent the source

architecture model and the target model, and M indicates the mapping relations;

Outputs: T (means consistent), F (means inconsistent);

Variables: DC (a Boolean variable);

1. DC:=F;
2. If $M(V_S)\not\subset V_T$ and $M(V_S)\neq V_T$, then the mapping is inconsistent; goto step 6;
3. Calculating the transitive closure of graph G_S, denoted as $Closure_S$;
4. Calculating the transitive closure of graph G_T, denoted as $Closure_T$;
5. if $M(Closure_S)\subseteq Closure_T$, then the mapping is consistent of component dependency and let DC:=T; otherwise the mapping is inconsistent;
6. return DC.

The time complexity of Algorithm 1 is the same as that of calculating transitive closure, which is $O(n^3)$.

Borrowing the concept of weak equivalence of PA[10], we can make a judgment on behavioral semantic consistency of model transformation according to the behavioral traces described in component specifications at different levels of abstraction. For example, the external behaviors of the two corresponding component models (respectively denoted as AM_S and AM_T) at different levels are respectively formalized by two processes BP_i and BP_j, then the mapping M: $AM_S \rightarrow AM_T$ is called preserving behavioral semantics if and only if BP_j is weakly equivalent to BP_i. It will be illustrated further by a practical case in the next section.

4 A Case Study

The supply chain management system shown in Fig.1 is still used in this section to illustrate the application of the theory and approach proposed in this paper. We assume that, in the target architecture after model transformation, the combination of three subcomponents (respectively named *ShopAgency*, *StoreAgency* and *BankAgency*) achieves the functions of the *Shop* in the source model. The *ShopAgency* is responsible for interacting with customers, and the *StoreAgency* is used for accessing to the stock data service and interacting with the component *Transport*, and the *BankAgency* is in charge of processing transactions with the banking services.

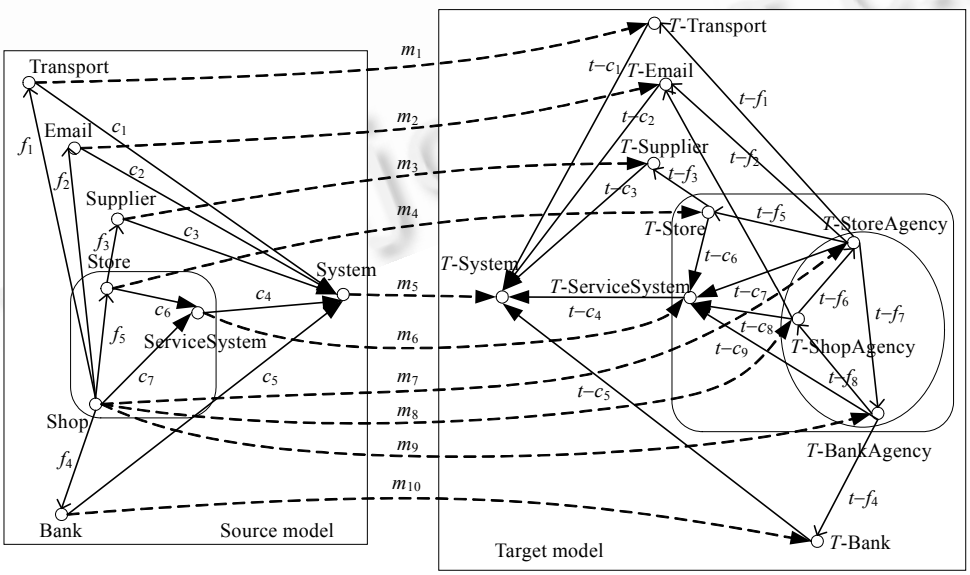

Fig.3 Mapping relations between the source model and the target

The categorical diagram of the source model is shown as the left part of Fig.3, where the morphisms c_1~c_7 are

composition morphisms, and f_1~f_5 are interaction morphisms. The corresponding target architecture model represented within a categorical diagram is shown as the right part of Fig.3. The mapping relations from the source to the target are drawn with dashed arrows, which satisfy the commutative law[8] of the category diagram, such as $t–f_1\circ m_7=m_1\circ f_1$, $t–c_4\circ m_6=m_5\circ c_4$, $t–f_5\circ m_7=m_4\circ f_5$, and so on. These properties show that the transformation following these mappings preserves consistency of the dependency relations among the components. Due to the limited space, the component specifications for the *ShopAgency*, the *StoreAgency* and the *BankAgency* are all omitted in this paper, and only their colimit specification computed according to Definition 5 and Definition 6, denoted by $CP_{T\text{-}Shop}$, is shown in Fig.4, where some inner port descriptions are left out.

```
Component T-Shop
  Attributes
      Private Cid: String;
      Private OrderedItem: ItemType;
      Private RecedeItem: ItemType;
      Private PayInfo: Boolean;
      Private ReStoreItem: ItemType;
      Private ChkAvail: Boolean;
      …
  Ports
      In PT1{
          Boolean SellItem(item: ItemType);
          Boolean Recede(item: ItemType);
      }
       Out PT2{
          Void SdLetter(letter: LetterType);
      }
       Out PT6{
          Void SdLetter(letter: LetterType);
      }
       In/Out PT7{
          Boolean ShipItem(item: ItemType);
          Boolean Withdraw(item: ItemType);
      }
       In/Out PT8{
          Boolean ChkStore(item: ItemType);
          Boolean RcoStore(item: ItemType);
      }
       In/Out PT12{
          Boolean ProcPay(payinfo: PayInfoType);
          Boolean Compensate(payinfo:PayInfoType);
          Boolean InfoBack(payinfo: PayInfoType);
      }
      …
  Axioms
      SellItem(item)⇒OrderedItem=item;
      Recede(item)⇒RecedeItem=item;
      RcoStReq(item)⇒ReStoreItem=item;
      PayInfo=InfoBack(payinfo);
      ChkAvail=ChkStore(item);
      InfmShip(item)⇒RecPayInfo(payinfo)=OK;
      SdLetter(letter)⇒RecPayInfo(payinfo)=FAIL;
      RcoStReq(item,0)⇒RecPayInfo(payinfo)=FAIL;
      RcoStReq(item,1)⇒RecPayInfo(payinfo)=OK;
      SdLetter(letter)⇒(ChkStReq(item)=FAIL)∨(RecPayInfo(payinfo)=FAIL);
      PayFeeReq(payinfo)⇒ChkStReq(item)=OK;
      ProcPay(fee)=FAIL⇒RcoStore(item);
      (ChkStore(item)=FAIL)∧(ProcPay(fee)=OK)=FALSE;
      ChkStore(item)=FAIL⇒Order(order);
      …
  Behavior
      BP_AG Δ= BuyRequest.TrnChkStore.(RecChkOK.TrnReqPay.(RecPayFeeOK.ShipItem.BP_AG+
               PayFailRec.RcoStore.SendLetter2.BP_AG)+RecChkFail.SendLetter1.BP_AG)+
               RecedeRequest.Withdraw.TrnCompensate.RcoStore.BP_AG;
  …
```

Fig.4 Colimit specification of the *ShopAgency*, the *StoreAgency* and the *BankAgency*

Next, we analyze the behavioral semantic descriptions BP_{Shop} in Fig.2 and BP_{AG} in Fig.4. Obviously, BP_{Shop} is weakly equivalent to BP_{AG}. According to the description in Section 3, we know that the transformation M: $M_{Shop} \rightarrow M_{T\text{-}Shop}$ consistently preserves behavioral semantics.

5 Conclusion and Future Work

In this paper, category theory and process algebra are combined together to provide a unified semantic description framework for architecture models and their mapping relations. Architecture is inherently about putting parts together to make larger systems. The colimit operation of category theory and the parallel composition operator of process algebra work particularly well in this regard. In this way, one can reason about all parts of a system separately, which preserves the properties established about the parts. The semantic description framework commendably captures the essence, process and requirements of MDD, which can be used as a new theoretical guidance for the cognition, design and semantic calculation of model transformations and model-driven development. As far as future work is concerned, there are several directions that we would like to explore: (1) to study more about the semantic properties which should be preserved in model transformations; (2) to make a summary of the generic proving processes and propose algorithms to strictly prove whether a transformation satisfies a property preservation constraint or not, and thus support the design and verification of transformation rules.

References:

[1] Hailpern B, Tarr P. Model-Driven development: The good, the bad, and the ugly. IBM Systems Journal, 2006,45(3):451−461.

[2] Varró D, Pataricza A. Automated formal verification of model transformations. In: Proc. of the UML 2003 Workshop on Critical Systems Development in UML. San Francisco, 2003. 63−78. http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2003/csduml2003_vp.pdf

[3] Thomas D. MDA: Revenge of the modelers or UML utopia? IEEE Software, 2004,21(3):15−17.

[4] Liu H, Ma ZY, Shao WZ. Description and proof of property preservation of model transformations. Journal of Software, 2007, 18(10):2369−2379 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/18/2369.htm

[5] Bergstein PL. Object-Preserving class transformations. In: Proc. of the ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications. New York: ACM Press, 1991. 299−313.

[6] Van Der Straeten R, Jonckers V, Mens T. Supporting model refactorings through behaviour inheritance consistencies. In: Proc. of the Unified Modeling Language (UML 2004). LNCS 3273, Heidelberg: Springer-Verlag, 2004. 305−319.

[7] Mens T, Van Eetvelde N, Demeyer S, Janssens D. Formalizing refactorings with graph transformations. Journal of Software Maintenance and Evolution: Research and Practice, 2005,17(4):247−276.

[8] Pierce Benjamin C. Basic Category Theory for Computer Scientists. Cambridge, Massachusetts: MIT Press, 1991.

[9] Lopes A, Wermelinger M, Fiadeiro JL. Higher-Order architectural connectors. ACM Trans. on Software Engineering and Methodology, 2003,12(1):64−104.

[10] Bernardo M, Ciancarini P, Donatiello L. Architecting families of software systems with process algebras. ACM Trans. on Software Engineering and Methodology, 2002,11(4):386−426.

[11] Lu RQ. Towards a mathematical theory of knowledge. Journal of Computer Science and Technology, 2005,20(6):751−757.

附中文参考文献:

[4] 刘辉,麻志毅,邵维忠.模型转换中的特性保持的描述与验证.软件学报,2007,18(10):2369−2379. http://www.jos.org.cn/1000-9825/18/2369.htm

HOU Jin-Kui was born in 1976. He is a Ph.D. candidate at the Computing Science from Shandong University. His main research areas are model-driven development, formal method and software architecture.

WANG Hai-Yang was born in 1965. He is a professor and doctoral supervisor at the Shandong University and a CCF senior member. His research areas are software and data engineering, CSCW and business process management.

MA Jun was born in 1956. He is a professor and doctoral supervisor at the Shandong University and a CCF senior member. His research areas are analysis and design of algorithms, information retrieval and parallel computing.

WAN Jian-Cheng was born in 1949. He is a professor and doctoral supervisor at the Shandong University and a CCF senior member. His research areas are software architecture, software engineering and natural language processing.

YANG Xiao was born in 1981. She is a Ph.D. candidate at the Computing Science from Shandong University. Her research areas are natural language processing, artificial intelligence and software engineering.