Formal Semantic Meanings of Architecture-Centric Model Mapping

Xiao Yang, Jinkui Hou, and Jiancheng Wan

School of Computer Science and Technology, Shandong University,
Jinan, 250061, China
{yangx,houjk}@mail.sdu.edu.cn, wanjch@sdu.edu.cn

Abstract. Over the past few years, Model-driven Development (MDD) has become an active research area of software engineering, in which model transformation is a key technology. However, there is currently no mature foundation on the definition of mapping rules as well as cardinal principles to verify the mapping relations between such models. Based on software architecture, category theory is used to explore the mapping relations between models at different abstract levels, so that the interconnections and mapping relations between component-based models and the compositions of these relations have rigorous meanings. The morphism composition and functors are used to trace the relationships between component models at different abstract levels. Formal description of model mappings is suitable to the automatic software development. It can be a measurement for validating the mapping rules between different models, and thus can make an effective support to MDD.

1 Introduction

Over the past few years, Model-driven Development (MDD) has become an active research area of software engineering [1], in which model transformation is a key technology. Represented by OMG's MDA, numerous research institutions and enterprises have been investing a large amount of money and manpower in the model transformation study. Currently, a number of products based on MDA have proved that a lot of benefits can be obtained from it, such as rapid development, architecture advantages, improvement of code consistency and maintainability, enhancement of system's portability across middleware vendors, and it also shows great potential in these areas.

Most of the existing and proposed approaches [2] for model transformation focus on providing a concrete solution for the transformation between models at different abstract levels, and there's currently no mature foundation on the definition of mapping rules as well as cardinal principles to validate the mapping relations between such models. More in-depth study and formal methods about this issue are expected to support complicated model transformation [3]. Consequently, a unifying framework for the mapping description techniques seems imperative. Such a framework should be formal, in order to avoid ambiguities; offer a sufficiently high

M. Xu et al. (Eds.): APPT 2007, LNCS 4847, pp. 640–649, 2007.
© Springer-Verlag Berlin Heidelberg 2007

level of abstraction, in order to concentrate on the meaning of concepts instead of on representational aspects; and be sufficiently expressive. These requirements suggest category theory as an excellent candidate.

Category theory is a mathematical framework suitable for representing relationships between knowledge [4], which has been viewed by the computer sciences as a means of achieving representation independence and abstraction, while providing conceptual subdiscipline unification [5]. Category theory has been widely used to facilitate specification construction [6]. It provides the right level of mathematical abstraction to address languages for describing software architectures [7]. The abstract framework of category theory is shown to provide semantics for the configuration of complex systems from their component parts. Diagrams express configuration by representing the results of applying combinators to recursively defined system components. These ideas are extended to provide a precise semantics for both components structuring and models mapping in this paper. We propose adaptations to the categorical framework in order to manage model mapping and transformation.

The rest of this paper is organized as follows: some basic concepts of category and algebraic specification are given briefly in Section 2; the formal semantic meanings of component model mapping are developed in Section 3; a case study about email client model mapping is shown in Section 4; related works in this area are presented in Section 5; The paper ends with conclusions and future works.

2 Category Theory and Algebraic Specification

In computing science, more abstract viewpoints are often more useful, because of the need to achieve independence from the overwhelmingly complex details of how things are represented or implemented. Category theory allows the study of the essence of certain concepts as it focuses on the properties of mathematical structures instead of on their representation. One of the basic principles summarized in [8] is that complex systems can be usefully identified with diagrams, system components and connectors corresponding to nodes, and interconnections being established through the edges of the diagrams. Category theory is ideal for this purpose, as it provides a rich body of theory for reasoning about objects and relations between them. Moreover, category theory lends itself well to automation, so that, for example, the composition of two specifications can be derived automatically, provided that the category of specifications obeys certain properties. Most of the category definitions of this section are adapted from [4].

Definition 1. Category. A category $\mathbb{C}$ is composed of two collections:

(1) the objects of the category, which is called C-objects;

(2) the morphisms (arrows) of the category, which is called C-arrows;

These two collections must respect the following properties:

(a) each morphism f is associated with an object A that is its domain and an object B that is its codomain. Notation: $f: A \rightarrow B$.

(b) for all morphisms $f: A \rightarrow B$ and $g: B \rightarrow C$, there exists a composed morphism $g \circ f: A \rightarrow C$ and the composition law is associative, i.e. for all $h: C \rightarrow D$, $h \circ (g \circ f) = (h \circ g) \circ f$.

(c) for each object A of the category, there exists an identity morphism id_A such that:

$$\forall f: B \rightarrow A,\ id_A \circ f = f \text{ and } \forall f: A \rightarrow B;\ f \circ id_A = f.$$

Many categorical definitions and proofs employ diagrams. As remarked before, quite complex facts can be visualized by the use of these diagrams.

Definition 2. Diagram. A diagram $\mathbb{D}$ in a category $\mathbb{C}$ consists of a collection D_C of C-objects and a collection D_A of C-arrows such that for any arrow $a \in D_A$, $cod\ a \in D_C$ and $dom\ a \in D_C$, where $cod\ a$ represents the codomain of a and $dom\ a$ represents the domain of a.

Definition 3. Commutative diagram. A diagram is said to *commute* if every path between two objects in its image determines through composition the same arrow. The case is shown in Fig.1.

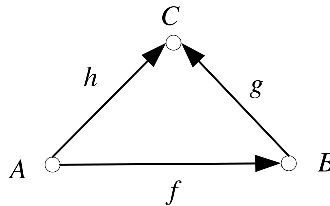

Fig. 1. Diagram commutes iff h is the composite g ∘ f

A powerful construction operation called colimit is defined over diagrams.

Definition 4. Colimit. A colimit for a diagram $\mathbb{D}$ in a category $\mathbb{C}$ is a C-object C along with a co-cone$\{f_i : D_i \rightarrow C \mid D_i \in \mathbb{D}\}$ from **D** to C such that for any other co-cone $\{f'_i: D_i \rightarrow c' \mid D_i \in \mathbb{D}\}$ from **D** to a vertex C', there is a unique C-arrow $f : C \rightarrow C'$ such that for every object D_i in $\mathbb{D}$, the diagram shown in Fig.2. commutes; i.e., $f \circ f_i = f'_i$.

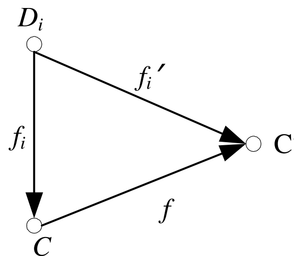

Fig. 2. Definition of a colimit

A practical interpretation for the colimit is given by Goguen in [5]: “Given a species of structure, say widgets, then the result of interconnecting a system of widgets to form a super-widget corresponds to taking the colimit of the diagram of widgets in which the morphisms show how they are interconnected.”

Definition 5. Functor. A functor F from a category $\mathbb{C}$ to a category $\mathbb{D}$ is a function which assigns to each C-object a, a D-object $F(a)$, and to each C-arrow $f: A \rightarrow B$, a D-arrow $F(f): F(A) \rightarrow F(B)$, such that identity arrows and composites are preserved, i.e., $F(id_A) = id_{F(A)}$; for all C-objects A, and $F(g \circ f) = F(g) \circ F(f)$; whenever $g \circ f$ is defined in $\mathbb{C}$.

Category theory can be used to compose formal specifications from smaller, reusable pieces. When used for specification construction, there is usually a requirement that the morphisms preserve theoremhood. That is, if a morphism between two specifications is defined, there is an obligation to prove that the axioms of the source specification are theorems of the target specification under the translation. Thus we can define an architecture model as a diagram of specifications, and prove properties of this architecture at a relatively abstract level.

3 Formal Semantic Meanings for Architecture Model

In MDD, the model description must be precise enough to grasp the essential behavior of the component, which also must be sufficiently abstract to ensure that, according to the requirements of the model, different vendors can respectively develop their component products that can compete with each other. A category theoretic foundation is shown in this section for the conceptual component modeling elements.

3.1 Component Signature and Component Specification

From a mathematical point of view, component signatures are structures defined as follows.

Definition 6. Component signature. A component signature is a 6-tuple $<\Sigma, A, \Gamma, fa, fp, D>$ where

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

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

(3) Γ is an S^*-indexed family of port symbols.

(4) $fa: A \rightarrow S_A$, $S_A \subset S$ is a total function, which shows the properties of the attribute;

(5) $fp: \Gamma \rightarrow S_T$, $S_T \subset S$ is a total function, which shows the properties of the ports;

(6) $D: \Gamma \rightarrow 2^A$ is a total function, for each $g \in \Gamma$, $D(g)$ is the collection of the attributes which can be modified via port g.

Definition 7. Component signature morphism. Given two component signatures $\theta_1 = <\Sigma_1, A_1, \Gamma_1, fa_1, fp_1, D_1>$ and $\theta_2 = <\Sigma_2, A_2, \Gamma_2, fa_2, fp_2, D_2>$, a morphism $\sigma: \theta_1 \rightarrow \theta_2$ from θ_1 to θ_2 consists of:

(1) a morphism of algebraic signatures $\sigma_v: \Sigma_1 \rightarrow \Sigma_2$;

(2) for each f: s_1, …, $s_n \rightarrow s$ in A_1, an attribute symbol $\sigma_a(f)$: $\sigma_v(s_1)$, …, $\sigma_v(s_n)$ $\rightarrow \sigma_v(s)$ in A_2;

(3) for each g: $s_1,\ldots,s_n$ in Γ_1, an action symbol $\sigma_\gamma(g)$: $\sigma_\gamma(s_1),\ldots,\sigma_\gamma(s_n)$ in Γ_2;

(4) for each $g \in \Gamma_1$, $\sigma_a(D_1(g))=D_2(\sigma_a(g))$.

The last conditions show that the attributes affected by a certain port must be preserved through a component signature morphism.

Definition 8. Component specification. A component specification CS is a pair (θ, Δ), where θ is a component signature $<\Sigma, A, \Gamma, fa, fp, D>$ and Δ, the body of the specification, is a quadruple (I, F, B, Φ), where

(1) I is a θ-proposition (constraining the initial values of the attributes);

(2) F assigns to every port $g \in \Gamma$ a non-deterministic command, i.e. F maps every attribute a in $D(g)$ to a set expression $F(a)$;

(3) B assigns to every port $g \in \Gamma$ a θ-proposition as its guard.

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

We distinguish between functional requirements and nonfunctional requirements. Functional requirements describe the system behavior as well as the collaboration among system components to accomplish the system behavior. nonfunctional requirements pertain to how a system performs its functions and include concerns such as quality, quantity, and timeliness.

Definition 9. Component specification morphism. A morphism ω: $CS_1 \rightarrow CS_2$ of component specification $CS_1 = <\theta_1, \Delta_1>$ and $CS_2 = <\theta_2, \Delta_2>$, consists of a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$ such that,

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

(2) $\exists g_1 \in \Gamma_1$, $a_1 \in D_l(g_l)$, $B_2(\sigma(g_1)) \supset \sigma(F_1(g_1, a_1)) = F_2(\sigma(g_1), \sigma(a_1))$;

(3) $I_2 \supset \omega(I_1)$.

(4) $\exists g_1 \in \Gamma_1$, $B_2(\sigma(g_1)) \supset \sigma(B_1(g_1))$.

Requirements shown above allow guards to be strengthened but not to be weakened.

3.2 Component Relations and The Hierarchy Component Models

Relationships between components impose accessibility constraints on their attributes and, thus, restrict the way components can be interconnected. In the component-based model-driven development [9], there are many kinds of relations between component models, such as *compose*, *use*, *extend*, as well as the *mapping* relations between component models at different abstract levels [10].

A specification morphism m: $A \rightarrow B$ from a specification A to specification B maps any element of the signature of A to an element of the signature of B that is compatible (i.e., sort with sort etc). The *compose* relationship express how that

component is part of the given ones. On this basis, a *compose* relation between two components S_1 and S_0 is achieved in category theory by identifying a morphisms c_1 from S_1 to S_0, which express that S_1 is a subcomponent of S_0. This case is expressed by Fig 3 (a). Through this morphism, the configuration diagram returns a new component that represents the overall system. Some constraints, however, apply. The *use* and *extends* relationship may describe a dependency between two implementations or between two specifications. It actually applies to different yet closely related component relationships. These dependency relationships between components at the same level are represented by the morphisms given in Fig. 3 (b) and (c), which shows that the implementations of some functions in R_1 (or C_2) are based on the functions specified in R_2 (or C_1). The *mapping* relations describe the key relationship between abstract component specifications and concrete component specifications. It is also formalized via morphisms in category theory. As shown in Fig.3. (d), the morphism between S and T is as an illustration, where T is the direct corresponding part to S at a more concrete level. The *mapping* relationship can be defined informally as follows: *Abstract component S* is mapped to *concrete component T* if and only if T exhibits the behavior specified by S.

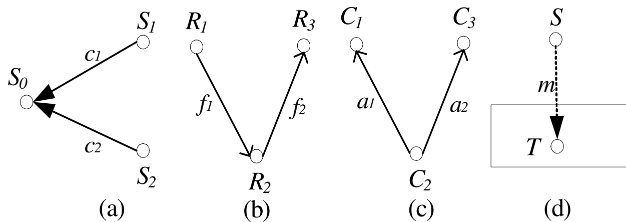

Fig. 3. Component specifications morphisms

The composition of component specifications can be modeled hierarchically in a category theoretic framework. Large complex systems are put together, or configured, from smaller parts, some of which have already been put together from even smaller parts. The composition operation then defines and constructs an aggregated component describing the overall system from the individual components and their interactions. Colimits can be used to construct systems from simpler components in our category of component models. We consider systems composed of a number of components coordinating their activities. The components of a system are represented by recursively defined objects and configured by combinators. Under such interpretation, a categorical diagram represents a system of components. A colimit of a diagram, if it exists, allows one to represent the whole system as a single component.

Properties can be associated to each specification. These are the properties that we expect the component to respect; that we need to prove on the component. We can represent these properties in the same framework as the specifications and this allows us to use category theory and particularly categorical computations to manage them. The property of the colimit specifications gives the composition for the properties. The advantage of this approach is that the management of properties and their status (proved, to be proved) is handled in a uniform way through the management of morphisms and proof obligations.

3.3 Architecture Models and Mapping Functors

Software architecture is a world populated by components, connectors, configurations, etc [6]. As a simple example, an architecture theory could be defined by the objects and the composition rules. These rules provide the semantics of the architecture, and can be used to both interpret the meaning of structures and to identify equivalent or included substructures. These notions can be formalized as a category.

Definition 10. Architecture Model. An architecture model is a 5-tuple $<CO, CR, OT, RT, \vdash>$, where

(1) CO is a collection of components;

(2) CR is a collection of binary relations defined over CO;

(3) OT is a collection of component specifications, and for every component $o \in CO$, $type(o) \in OT$, herein the operation *type* returns the component's type;

(4) RT is a collection of binary relation types defined over OT, for each $r \in CR$, $type(r) \in RT$;

(5) $\vdash$ is a satisfaction relation between OT-sentences and OT-models [10] such that $\vdash$ defines a well order.

Component specifications and cs-morphisms constitute a category for architecture model, henceforth denoted by AM. Obviously, the category AM is cocomplete.

Category theory also provides us with the means to establish relationships between different architectural models: functors. An architecture mapping from AM_1 to AM_2 is simply a mapping of component specifications of AM_1 to component specifications of AM_2 that preserves relations between these components.

Definition 11. Architecture mapping functor. An architecture mapping functor denoted $F: AM_1 \rightarrow AM_2$ from architecture model $AM_1=<CO_1, CR_1, OT_1, RT_1, \vdash>$ to $AM_2=< CO_2, CR_2, OT_2, RT_2, \vdash>$ is a function $F: AM_1 \rightarrow AM_2$, in such a way that

(1) for every component $o \in CO_1$, $F(o) \in CO_2$;

(2) for every component specification o, $o' \in CO_1$, $o \rightarrow o' \in CR_1$ implies $F(o) \rightarrow F(o') \in CR_2$;

(3) $F(f \circ g)=F(f) \circ F(g)$; whenever $g, f \in CR_1$ and $f \circ g$ is defined;

(4) for all OT_1 sentence s, $AM_1 \vdash s$ if and only if $AM_2 \vdash F(s)$.

According to the theory of model-driven development [11], a mapping functor between two architecture models at different abstract levels for the same system is a mapping in case the axioms of the source are logically implied by the axioms of the target under the translation. Thus, architecture mapping preserve the properties of the source architecture models. Functors map the objects and morphisms of one category to corresponding objects and morphisms of another category. Consistency between the sorts and operations of the component specifications are maintained.

4 A Case Study

In this section, a component-based model for email client was used as a simple case to illustrate the feasibility of the approach proposed in this paper.

Based on hierarchy component model, the structure of the source model was depicted within categorical diagram in the left part of Fig.4., which involving seven components types: (1) *MainUI* is in charge of the UI layout and art design of the interaction between the mail client and users, through which users can receive email, check email, send email and compose email; (2) *EmailManagement* is responsible for the storage, reading and display of all the e-mails stored locally; (3) *Editor* is used to composing text format or html format e-mail; (4) *Client* is responsible for sending and receiving e-mail; (5) *Protocol* is identified for the setting of mail protocols; (6) *AddressBook* is used for the management of address book; (7) *Account* is responsible for account management. Herein, *EmailManagement* and *Editor* are two composite components. In the component *EmailManagement*, a general-used component named *GeneralList* used to handle mail-lists, a *DataAccess* component used to access email information, and a component *FileView* used to show details of different kinds of e-mails as well as the corresponding component *ManageUI* for user interface were introduced as four sub-components. Herein, the component *FileView* was composed of three sub-components: *HtmlView* to deal with Html format documents, component, *TextView* to manage text format documents and *MultimediaView* to process multimedia documents. In the component *Editor*, the sub-component *EditorUI* is responsible for the user interface, and two subcomponents named *TextEdit* and *HtmlEdit* respectively are used to compose different formats emails.

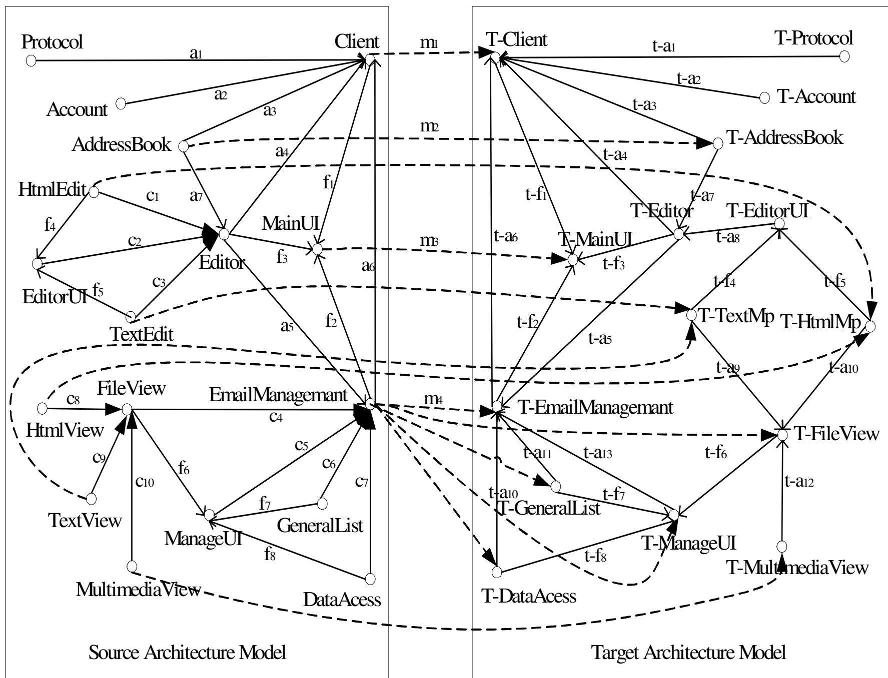

Fig. 4. Component model mapping of email client

We assume that the target platform does not support composite components, such as the programming language C++ does not support nested definition of the Class, and the EJB specification, only permits several javabeans be included in a jar package, but do not support the definition of composite EJB. In such case, the mapping relations between the majority of the source atomic components and the components types of the target platform can easily built. As for the composite components, the decomposition mapping relations must be built through stepwise layers-decomposition. In order to optimize the target architecture, there is generally a need to integrate target components specifications, and thus will form the composition mapping from the source model to the target model. In this case, the functions for html document editing and browsing were combined into a component *T-HtmlMp* in the target architecture. Similarly, the functions for text document editing and browsing were combined into a component *T-TextMp*.

The corresponding target architecture model was represented within categorical diagram in the right part of Fig.4. The mapping relations from the source to the target can be observed by component names. Only a part of mapping morphisms are drawn in Fig. 4, which satisfy the commutative law of category diagram, such as $t\text{-}f_1 \circ m_1 = m_3 \circ f_1$, $tf_3 \circ t\text{-}a_7 \circ m_2 = m_3 \circ f_3 \circ a_7$, and so on. This property shows that the transformation following these mappings persevere consistency of dependency relations among the components.

5 Related Work

In the past few years, a large number of approaches for model transformation have been proposed. Most of these approaches lay emphasis on providing a concrete solution for the transformation from source model to target model. In the work by Bezivin et al [12], the impact on the efforts to define mapping rules caused by the gap between the source and the target modeling languages is mentioned briefly from the view of meta-model semantics, but no general solutions are given. The central role of formalism extension mechanisms in managing the abstraction-level gap between modeling languages as well as the platform-level details of specific implementations is shown in Caplat and Sourrouille's work [3]. The gap between the modeling languages can be narrowed using this mechanism, but cannot be completely eliminated. The mapping relations between models are still difficult to define directly. On the other hand, category theory has been widely used to facilitate specification construction. In Gerken's work [6], category theory and algebraic specifications were used to develop a formal definition of architecture and it also showed how architecture theory can be used in the construction of software specifications. The problem of interconnection relationships in large systems was addressed using category theory in Guo's paper [10], which also gives a framework of the dependencies modeling. The work by Fiadeiro and Maibaum [7] have showed how elementary concepts of category theory can be used to formalize key notions of software architecture independently of the formalism chosen for describing the behavior of components. Despite the popularity of category theory in specification construction, little attention was given to understanding the relationship between levels of abstraction for component-based model mapping.

6 Conclusion and Future Work

In this paper, a unifying framework for component-based model mapping was presented. The framework is based on category theory due to its formality and its high level of abstraction. An important contribution is the formalization of mappings between architecture modes at different abstract levels. In order to specify and verify such a model mapping to ensure semantic compatibility, we postulate that through formality, the terms “component” and “architecture” both can be precisely defined and some important properties of systems can be investigated with precision. Furthermore, we use category theory to develop a formal definition of architecture mapping and some important properties are exploited. It can be a measurement for validating the mapping rules between models at different abstract levels, and thus to provide an effective support to model driven software development.

Future works are as follows: (1) further to formalize the definition of component specification and architecture model, and thus to strengthen the abilities of semantic expressiveness and consistent verification between models; (2) more study about the preserving of semantics features in model mapping for the enhancement of accuracy.

References

1. Brent, H., Peri, T.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal 45(3), 451–461 (2006)
2. Krzysztof, C., Simon, H.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–644 (2006)
3. Caplat, G., Sourrouille, J.L.: Model Mapping Using Formalism Extensions. IEEE Software 22(2), 44–51 (2005)
4. Barr, M., Wells, C.: Category Theory for Computing Scince. Prentice-Hall, Englewood Cliffs (1990)
5. Goguen, J.: A Categorical Manifesto. Mathematical Structures in Computer Science 1(1), 49–67 (1991)
6. Mark, J.G.: Specification of Software Architecture. Journal on Software Engineering and Knowledge Engineering 10(1), 69–95 (2000)
7. Fiadeiro, J.L., Maibaum, T.: A Mathematical Toolbox for the Software Architect. In: Proc. 8th International Workshop on Software Specification and Design, pp. 46–55 (1995)
8. Eilenberg, S., MacLane, S.: General theory of natural equivalences. Transactions of the American Mathematical Society 58(1), 231–245 (1945)
9. Colin, A., Joachim, B., Christian, B., et al.: Component-Based Product Line Engineering with UML, Addison-Wesley Professional, Pearson Education, Boston (2002)
10. Guo, J.: Using category theory to model software component dependencies. In: ECBS 2002. Proc. of the 9th Annual IEEE Int’l Conf. and Workshop on the Engineering of Computer-Based Systems, pp. 185–192. IEEE Computer Society, Los Alamitos (2002)
11. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston (2003)
12. Bezivin, J., Hammoudi, S., Lopes, D., Jouault, F.: Applying MDA approach for Web service platform. In: Proc. of Enterprise Distributed Object Computing Conference, Monterey, California, USA, pp. 58–70 (2004)