Available online at www.sciencedirect.com
SCIENCE DIRECT®

ELSEVIER Electronic Notes in Theoretical Computer Science 122 (2005) 229–245

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Coalgebraic Semantic Framework for Component-based Development in UML

Sun Meng[a,1] , Bernhard K. Aichernig[b], Luís S. Barbosa[c,2] and Zhang Naixiao[a,1]

[a] *LMAM, School of Mathematical Science, Peking University, Beijing, China*

[b] *International Institute for Software Technology, The United Nations University*

[c] *Department of Informatics, Minho University, Braga, Portugal*

Abstract

This paper introduces a generic semantic framework for component-based development, expressed in the unified modelling language UML. The principles of a coalgebraic semantics for class, object and statechart diagrams as well as for use cases, are developed. It is also discussed how to formalize the refinement steps in the development process based upon a suitable notion of behavior refinement. In this way, a formal basis for component-based development in UML is studied, which allows the construction of more complex and specific systems from independent components.

Keywords: Unified modeling language, refinement, UML, coalgebras

1 Introduction

Component-based development [22] became accepted in industry as a new effective paradigm and widely considered the cornerstone of software engineering in the years to come. Within the component-based paradigm a system consists of a collection of components. Each component is a unit of composition

[1] This piece of research is partially supported by the National Natural Science Foundation of China under grant no. 60273001.

[2] The work of Luís S. Barbosa is supported by FCT, under contract POSI/ICHS/44304/2002, in the context of the PURe project.

doi:10.1016/j.entcs.2004.06.051

with contractually specified interfaces. It can be deployed independently and is subject to composition by third parties.

It has been widely recognized that component-based development is a rather intricate activity and there is still no general agreement on precise definitions of both components and component assembly. Thus the need for formal foundations of component-based development was identified, especially in relation to widely used component-oriented design notations such as UML [20].

The starting point of the research reported here is the observation that coalgebraic structures [11,21] may provide not only a powerful semantic domain for componentware, but also handy techniques for defining and reasoning about system's behavior, including a general notion of (bi)simulation and a coinduction proof principle. Therefore, this paper aims to a semantic framework in which the most popular notations used in UML can be mapped, in order to provide a basis for rigorous component-based development. The paper begins with a brief glimpse of the "components as coalgebras" approach, proposed by the authors in a series of previous papers [1,2,3]. This is followed, in Section 3, by a coalgebraic characterization of some crucial diagrams used in UML. In Section 4, the author's previous work on refinement of coalgebraic models discussed in [15], is framed to entail appropriate notions of refinement of UML models. This enables not only the formalization of (parts of) UML descriptions but also of its transformations as a basis for stepwise separate development. Finally, the prospects for future work are discussed in Section 5.

2 Components as Coalgebras

In this section we introduce a coalgebraic model for state-based components, following closely the "components as coalgebras" approach proposed by L. Barbosa *et al* in [1,2,3]. This approach provides an observational semantics for software components and a generic assembly calculus, in the sense that the proposed constructions are parametric on a notion of component behavior.

Components interact with their environment via interfaces. Every interface provides a set of typed channels for receiving and sending messages, acting as a type for the corresponding component.

Definition 2.1 Let I and O be sets of typed input and output channels, respectively. The pair (I, O), denoted by $I \blacktriangleright O$, is called an *interface* and any component p with such an interface is typed as $p : I \to O$.

In the simplest, deterministic case, the behavior of a component p is captured by the output it produces, which is determined by the supplied input. But reality is often more complicated, for one may have to deal with components whose behavioral pattern is, *e.g.*, partial or even non deterministic. Therefore, to proceed in a generic way, the behavior model is abstracted to a strong monad B. Of course, $\mathsf{B} = \mathsf{Id}$ retrieves the simple deterministic behavior, whereas $\mathsf{B} = \mathcal{P}$ or $\mathsf{B} = \mathsf{Id} + \mathbf{1}$ would model non deterministic or partial behavior, respectively. Therefore, a component $p : I \longrightarrow O$ can be modelled by a coalgebra for the **Set** endo-functor $\mathsf{T}^{\mathsf{B}} = \mathsf{B}(\mathsf{Id} \times O)^I$. In this way, its computation will not simply produce an output and a continuation state, but a B-structure of such pairs. Formally, such a component p is modelled as a pointed concrete coalgebra

$$\langle U_p, \overline{\alpha}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^I, u_0 \in U_p \rangle$$

where a specific value u_0 is taken as its 'initial state' (or 'seed') and the dynamics is captured by currying the state-transition function $\alpha_p : U_p \times I \longrightarrow \mathsf{B}(U_p \times O)$.

Successive observations of a component p reveal its allowed behavioral patterns. For each state value $u \in U_p$, the behavior of p at u (more precisely, from u onwards) organize itself into a tree-like structure, because it depends on the sequences of input items processed. Such trees, whose arcs are labelled with I values and nodes with O values, can be represented by functions from non empty sequence of I to B-structures of output items. In other words, the space of behaviors of a component with interface $I \blacktriangleright O$ is the set $(\mathsf{B}O)^{I^+}$, which is in fact the carrier ν_{T} of the final T^{B}-coalgebra $(\nu_{\mathsf{T}}, \omega_{\mathsf{T}} : \nu_{\mathsf{T}} \longrightarrow \mathsf{T}^{\mathsf{B}}\nu_{\mathsf{T}})$. Therefore, by finality, from any other T^{B}-coalgebra p, there is a unique morphism $[\![p]\!]$ making the following diagram to commute:

$$\begin{tikzcd}
\nu_{\mathsf{T}} \arrow[r, "\omega_{\mathsf{T}}"] & \mathsf{B}(\nu_{\mathsf{T}} \times O)^I \\
U_p \arrow[u, "{[\![p]\!]}"] \arrow[r, "\overline{\alpha}_p"] & \mathsf{B}(U_p \times O)^I \arrow[u, "{\mathsf{B}([\![p]\!] \times O)^I}"']
\end{tikzcd}$$

Applying morphism $[\![p]\!]$ to a state value $u \in U_p$ gives the observable behavior of a sequence of p transitions starting at u. By instantiating B with concrete strong monads like $\mathcal{P}$ and $\mathsf{Id} + \mathbf{1}$, it is possible to model different behavior patterns such as non-determinism and partial behavior respectively.

Having defined generic components as concrete coalgebras, one may naturally wonder how do they get composed and what kind of calculus emerges from this framework. Components are "arrows" and so arrows between components are "arrows between arrows", which motivates the adoption of a bicategorical framework to structure our semantic universe. In brief, a bicategory $\mathbf{Cp}$ is built whose objects are the interface universes $I, O, \cdots$, arrows are seeded concrete coalgebras and 2-cells (arrows between arrows) are the corresponding coalgebra morphisms. 2-cell composition is inherited from $\mathbf{Set}$ and the identity on component p is defined as the identity id_{U_p} on the carrier of p. Then for each pair $\langle I, O\rangle$ of objects, the construction of $\mathbf{Cp}$ defines a hom-category $\mathbf{Cp}(I, O)$, whose arrows

$$h : \langle U_p, \overline{\alpha}_p : U_p \to \mathsf{B}(U_p \times O)^I, u_p \in U_p\rangle \to \langle U_q, \overline{\alpha}_q : U_q \to \mathsf{B}(U_q \times O)^I, u_q \in U_q\rangle$$

are maps $h : U_p \to U_q$ that satisfy the following conditions:

$$\overline{\alpha}_q \cdot h = \mathsf{T}^{\mathsf{B}} h \cdot \overline{\alpha}_p \quad \text{and} \quad h u_p = u_q$$

For each triple of objects $\langle I, K, O\rangle$, a composition law is given by a functor:

$$;_{I,K,O} : \mathbf{Cp}(I, K) \times \mathbf{Cp}(K, O) \to \mathbf{Cp}(I, O)$$

whose action on components p and q is given by

$$p; q = \langle U_p \times U_q, \overline{\alpha}_{p;q}, \langle u_p, u_q\rangle \in U_p \times U_q\rangle$$

where $\alpha_{p;q} : U_p \times U_q \times I \to \mathsf{B}(U_p \times U_q \times O)$ is detailed as follows[3]:

$$\begin{aligned}
\alpha_{p;q} : U_p \times U_q \times I &\xrightarrow{\cong} U_p \times I \times U_q \xrightarrow{\alpha_p \times \mathsf{id}} \mathsf{B}(U_p \times K) \times U_q \\
&\xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\cong} \mathsf{B}(U_p \times (U_q \times K)) \\
&\xrightarrow{\mathsf{B}(\mathsf{id} \times \alpha_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O)) \\
&\xrightarrow{\cong} \mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O)
\end{aligned}$$

The action of ";" on 2-cells is simply given by $h; k = h \times k$. Finally, for each object K, an identity law is given by a functor $\mathsf{copy}_K : \mathbf{1} \to \mathbf{Cp}(K, K)$, whose action on objects is the constant component $\langle \mathbf{1}, \overline{\alpha}_{\mathsf{copy}_K}, * \in \mathbf{1}\rangle$, where $\alpha_{\mathsf{copy}_K} = \eta_{\mathbf{1} \times K}$. Similarly, the action on morphisms is the constant morphism $\mathsf{id}_{\mathbf{1}}$.

Besides the 'pipeline' sequential composition defined above, components can be aggregated in a number of different ways. A calculus emerges in [2] from the structure of $\mathbf{Cp}$ including a 'wrapping' combinator as well as combinators capturing *external choice* $p \boxplus q$, *parallel composition* $p \boxtimes q$ and *concurrent composition* $p \boxast q$. Finally, generalized interaction is catered through a sort of 'feedback' mechanism on a subset of input ends.

[3] Notice that μ is the monad multiplication, η its unit and τ_r and τ_l the right and left strengths, respectively.

3 A Formal Semantics for UML

The central idea of our research on the formalization of UML is that a set of UML diagrams denote coalgebraic specifications as introduced by Jacobs [8,9]. More precisely, we intend to translate all the graphical symbols and annotations in the UML syntax, into functors and properties of a coalgebraic specification. Therefore standard definitions in universal coalgebra, such as bisimilarity and refinement, become available to reason and transform UML designs.

3.1 Class Diagrams

One of the main components in a UML description (as presented in [17]) is the class diagram. A class diagram shows the static structure of a system, consisting of a set of classes and relationships between them.

In UML, a class is an abstract description of a set of objects with similar structure, behavior and relationships. The description of a class includes the attributes and operations common to all of the objects belonging to it. Every object o of a class $\mathbf{C}$ in a system has an identifier id_o which is unique within the system. We denote the set of all identifiers by Id. Therefore, an object $o : \mathbf{C}$ is represented as a triple $o = (id_o, U_\mathbf{C}, \alpha_\mathbf{C} : U_\mathbf{C} \to \mathsf{T}(U_\mathbf{C}))$ where $U_\mathbf{C}$ is the state space of class $\mathbf{C}$, which consists of all the possible states for a $\mathbf{C}$-object, and T is a functor encapsulating a signature of attributes and methods. During the lifetime of an object, its local state u may change over the state space $U_\mathbf{C}$, but its identifier id_o, $U_\mathbf{C}$ itself and the transition structure $\alpha_\mathbf{C}$ remain the same.

Following the work of Jacobs *et al* [7,8,19] on coalgebraic semantics, every class in a UML class diagram is taken as a coalgebraic specification **Spec**, *i.e.*,

Definition 3.1 A coalgebraic specification **Spec** is a tuple (T, Φ, Ψ) in which:

- T is a functor on a local state space U, representing the signature of all the attributes and methods of the class;
- Φ is a set of axioms giving the constraints on the functors for the attributes and methods to characterize the properties of the class;
- Ψ is an axiom describing the properties that hold for newly created objects.

A model (class implementation) of a given class specification **Spec** $= (\mathsf{T}, \Phi, \Psi)$ is a triple $c = (U, \alpha : U \to \mathsf{T}(U), u_0)$, where U is a carrier set interpreting the state space of the class, $\alpha : U \to \mathsf{T}(U)$ is the transition structure which satisfies all the properties given by Φ and $u_0 \in U$ is an initial state satisfying Ψ.

The semantics of a concrete class $\mathbf{C}$ in a UML class diagram is defined as the category $\mathbf{Coalg}(\mathbf{Spec})$, whose objects are models of the corresponding coalgebraic class specification **Spec** and the arrows are initial state preserving homomorphisms between them. Formally,

$$[\![\mathbf{C}]\!] \stackrel{\Delta}{=} \mathbf{Coalg}(\mathsf{T}_{\mathbf{C}}, \Phi_{\mathbf{C}}, \Psi_{\mathbf{C}}) \qquad \text{if } \mathit{isAbstract}(\mathbf{C}) = False$$

where $(\mathsf{T}_{\mathbf{C}}, \Phi_{\mathbf{C}}, \Psi_{\mathbf{C}})$ is the specification of class **C** and the boolean value function *isAbstract* specifies whether the class **C** can be directly instantiated. This category is a subcategory of **Cp**, defined in the previous section. In detail, its objects are

$$\mathit{Obj}(\mathbf{Coalg}(\mathsf{T}_{\mathbf{C}}, \Phi_{\mathbf{C}}, \Psi_{\mathbf{C}})) = \{c = (U_{\mathbf{C}}, \alpha_{\mathbf{C}} : U_{\mathbf{C}} \to \mathsf{T}_{\mathbf{C}}(U_{\mathbf{C}}), u_0 \in U_{\mathbf{C}}) \mid \\ (c \models \Phi_{\mathbf{C}}) \wedge (c, u_0 \models \Psi_{\mathbf{C}})\}$$

where $c \models \Phi_{\mathbf{C}}$ means that all the axioms in $\Phi_{\mathbf{C}}$ are satisfied by coalgebra c and $c, u_0 \models \Psi_{\mathbf{C}}$ means that the properties in $\Psi_{\mathbf{C}}$ are satisfied by the initial state u_0 of c. Their formalization is immaterial for the purposes of the present paper.

Let us now look at the specification of attributes. The default UML syntax is

$$\mathit{visibility\ name\!:\ type\text{-}expr}[\mathit{multiplicity\ ordering}] = \mathit{initial\ value}\{\mathit{property\text{-}string}\}$$

Therefore, the semantic function of an attribute *At* in class **C** is defined as follows:

$$[\![v\ At:\ T\ [m] = i\{p\}]\!] \stackrel{\Delta}{=} \{At : U_{\mathbf{C}} \to \mathsf{T}_{At}(U_{\mathbf{C}}) \mid [\![At[m]]\!] \wedge [\![At = i]\!] \wedge [\![At\{p\}]\!]\}$$

where $\mathsf{T}_{At} = \mathcal{P}[\![T]\!]$, with $\mathcal{P}$ standing for the powerset functor, which is required to model attribute's multiplicity (it can be dropped whenever the multiplicity is exactly one). The semantics for *multiplicity* of an attribute *At* in class **C** is

$$[\![At[m]]\!] \stackrel{\Delta}{=} \ \forall u \in U_{\mathbf{C}} \,.\, card(At(u)) = m$$

or, if specified as a range,

$$[\![At[l..k]]\!] \stackrel{\Delta}{=} \ \forall u \in U_{\mathbf{C}} \,.\, l \leq card(At(u)) \leq k$$

The *initial value* is used for initializing the attribute of a newly created object. Hence,

$$[\![At = i]\!] \stackrel{\Delta}{=} \forall (U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0) \in \mathit{Obj}([\![\mathbf{C}]\!]).At(u_0) = i$$

The optional *property string* indicates property values of the attribute, like changeability. Due to length restrictions, we omit its semantics as well as the

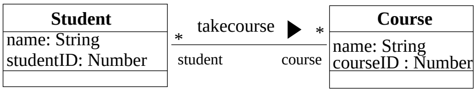

Fig. 1. Association

semantic functions for visibility of attributes, operations, interfaces, etc. which can be found in [14]. Note that in this framework attributes and operations are very similar: only the form of the signature functor changes. The distinction lies in the difference between observation and computation, only the latter being the source of change in object states.

An *association* in a class diagram describes the connections among objects in a system. It may have two or more association ends. Figure 1 shows an example of an association.

Suppose $\mathbf{Spec_U}$ and $\mathbf{Spec_V}$ are the class specifications of classes $\mathbf{U}$ and $\mathbf{V}$ in a class diagram and $\mathbf{A}$ is a binary association between them. Also assume that $c = (U, \alpha, u_0)$ and $d = (V, \beta, v_0)$ are objects in $\mathbf{Coalg}(\mathbf{Spec_U})$ and $\mathbf{Coalg}(\mathbf{Spec_V})$ respectively. Then association $\mathbf{A}$, which connects the two coalgebras, can be interpreted [4] as a space $S_A \subseteq \mathcal{P}((Id \times U) \times (Id \times V))$. Identifiers in the set Id are necessary to distinguish objects of the same class being in the same state. An element $s \in S_A$ is a state of the association which records a set of object pairs linked by the association simultaneously at the individual states paired at s. Every pair of objects in S_A is called a link between them.

Every association has three basic components: a name, a role and the multiplicity at each of its ends. The semantics for an association is given by the corresponding observers in each of the classes being related by the association. Let $\mathbf{A}$ be such an association between classes $\mathbf{U}$ and $\mathbf{V}$. The role names and multiplicities on the two ends are, respectively, u_A, v_A, and m_U, m_V, the later being two sets of non-negative integers. Then the semantics of $\mathbf{A}$ is defined as a pair of the coalgebraic observers:

$$(u_A : (Id \times V) \to \mathcal{P}(Id \times U),\ v_A : (Id \times U) \to \mathcal{P}(Id \times V)) \tag{1}$$

where U and V are the statespaces of coalgebras $(U, \alpha : U \to \mathsf{T}_U(U), u_0)$ and $(V, \beta : V \to \mathsf{T}_V(V), v_0)$ corresponding to the semantics of classes $\mathbf{U}$ and $\mathbf{V}$. Furthermore, the two coalgebraic observers must be related as follows:

$$o_U \in u_A(o_V) \Leftrightarrow o_V \in v_A(o_U) \tag{2}$$

[4] strictly speaking, what is connected by the association $\mathbf{A}$ are their classes: and the two coalgebras are connected by a link of $\mathbf{A}$.

Summing up, the semantics of an association is given by the pair of observers in (1) which satisfies (2):

$$[\![\mathbf{U}^{u_A} \underset{\mathbf{A}}{\underline{\quad}}^{v_A} \mathbf{V}]\!] \triangleq \{(u_A, v_A) \mid (2)\}$$

The UML specification [17] states that each association end has a multiplicity constraint (may be unspecified in an incomplete model) which is "a subset of the open set of non-negative integers". Should multiplicity be given explicitly in the diagram, the semantic function will become:

$$\begin{aligned}[\![\mathbf{U}^{u_A}_{m_U} \underset{\mathbf{A}}{\overline{\quad}}{}^{v_A}_{m_V} \mathbf{V}]\!] \triangleq \{(u_A, v_A) \mid (2) \wedge (\forall o_U : Id \times U.(card(v_A(o_U)) \in m_V)) \wedge \\ (\forall o_V : Id \times V.(card(u_A(o_V)) \in m_U))\}\end{aligned}$$

There is an entire family of optional properties that is provided by UML and may be given to associations in a class diagram. For example, UML allows an association to have its own attributes, which is represented by an association class, *i.e.*, an association which is also a class. Generally, an association class can be represented as a class with two one-to-many associations.

Therefore, the semantics of an *association class* is defined by the tupling of the semantics of this class together with the semantics of the two associations, as follows:

$$\begin{aligned}[\![\mathbf{U}^{u_V}_{m_U} \underset{\mathbf{AC}}{\overline{\quad}}{}^{v_U}_{m_V} \mathbf{V}]\!] \triangleq \{(ac, (u_A, a_U), (a_V, v_A)) \mid ac \in Obj([\![\mathbf{AC}]\!]) \wedge \\ (u_A, a_U) \in [\![\mathbf{U}^{u_A}_{1} -^{a_U}_{m_V} \mathbf{AC}]\!] \wedge (a_V, v_A) \in [\![\mathbf{AC}^{a_V}_{m_U} -^{v_A}_{1} \mathbf{V}]\!] \wedge \\ \forall o_U : Id \times U, o_V : Id \times V.((o_V \in v_U(o_U) \Leftrightarrow \\ \exists! o_A : Id \times AC.o_V = v_A(o_A) \wedge o_A = a_U(o_U)) \wedge \\ (o_U \in u_V(o_V) \Leftrightarrow \exists! o_A : Id \times AC.o_U = u_A(o_A) \wedge o_A = a_V(o_V)))\}\end{aligned}$$

The last predicate ensures the wellformedness of the object links via **AC** in both directions. Let us now turn to the semantics of *generalization.*

Generalization in a class diagram describes the inheritance relationship between a general class (superclass) and a more specialized class (subclass). The fact that a class **D** is a subclass of **C** in a class diagram is represented as **C** $\lhd\!\!-\!\!-$ **D**. It is also said that **D** inherits from **C**.

If there is such an inheritance relationship between **D** and **C**, a forgetful functor $G : \mathbf{Coalg}(\mathbf{D}) \to \mathbf{Coalg}(\mathbf{C})$ between the corresponding categories of models can be derived as shown in [7]. In our framework, the semantics of the generalization relationship in UML class diagrams is given by the collection of all the possible inheritance morphisms between the models of the corresponding class specifications, *i.e.*,

$$[\![\mathbf{C} \longleftarrow\!\!\!\!\lhd\; \mathbf{D}]\!] \triangleq \{g : d \to c \mid d \in \mathbf{Coalg}(\mathbf{D}) \wedge c \in \mathbf{Coalg}(\mathbf{C})\}$$

where g is the inheritance morphism from d to c. Such a morphism consists of a forgetful functor and a natural transformation between the functors corresponding to the two classes.

Generalization relations organize classes into a lattice, with the most generalized class at the top of the hierarchy (eventually an abstract class). The meet and join operators are defined as the superclass and subclass (for multiple inheritance) of classes respectively. Note that an abstract class may not have direct instances, and, therefore, it can not be interpreted in the same way as a concrete class. However, from the generalization relationship between an abstract class and its subclasses, its semantics can be recovered as the smallest superclass of all of its subclasses (or the least upper bound in the lattice of classes). Translated to category theory this means that the semantics of an abstract class with respect to its subclasses is the colimit of the corresponding subclass coalgebras, *i.e.*,

$$[\![\mathbf{C}\{\mathit{Abstract}\} \lhd\!\!\!\longleftarrow {}^{*}\{\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C_n}\}]\!] \triangleq \mathbf{Colimit_{Cp}}(c_1, c_2, \ldots, c_n)$$

where c_i are the coalgebras in $[\![\mathbf{C}_i]\!]$ respectively.

3.2 Object Diagrams and Class Diagrams

We may now sum up the previous discussion, centered at the class level, in order to give the semantics of *class diagrams*, which is defined via object diagrams.

An object diagram is a snapshot of the corresponding class diagram. It exhibits a set of objects and their relationships in a system at a specific point in time. These objects and their relationships are semantically defined as above. Thus, denoting the system state space by Σ, an object diagram represents a system state $\sigma \in \Sigma$ and can be seen as an instance of the corresponding class diagram. Therefore, an element $\sigma \in \Sigma$ is interpreted as the product of states of different objects at the same point in time.

Then, the system can be modelled as a coalgebra $c = (\Sigma, \alpha : \Sigma \to \mathsf{T}(\Sigma))$, where α describes all the possible transitions between system states. Formally,

Definition 3.2 The semantics of a class diagram **CD** is defined by the category $\mathbf{Coalg}(\mathbf{CD})$, whose objects are coalgebras $(\Sigma, c : \Sigma \to \mathsf{T}(\Sigma))$, where Σ is the system state space which contains all the possible system states. T is the tensor product of all the signature functors of the component classes and associations in the class diagram, which describes the possible system state transitions and observations. The arrows are T-homomorphisms between them.

3.3 Use Cases

In UML, *use cases* describe functional requirements. A use case is defined as “a sequence of transactions in a system, whose task is to yield a measurable value to an individual actor of the system” [12]. Thus, we can interpret a use case coalgebraically as a sequence of actions followed by some (eventually combined) observations. Single actions represent atomic use cases which change the system from one state to another. Such actions include creating new objects and deleting old objects, forming or deleting links between objects or modifying attribute values of objects.

The signature of an atomic use case is again defined by a functor T. Let Σ be the system state space, as defined in the semantics of class diagrams. Then a use case is interpreted as a function $uc : \Sigma \to \mathsf{T}(\Sigma)$. An example should demonstrate the coinductive formalization of use cases.

Consider a use case **BB** describing the process of borrowing a book in a library (see [14] for the complete example) as specified in Figure 2.

Let Σ be the state space of the system. Then the behavior of **BB** can be defined by a coalgebraic function $bb : \Sigma \to \Sigma^{Student \times Book}$. If student s_0 borrows book b_0, then the observable effects can be specified coinductively:

$$\begin{aligned} &\mathbf{pre} \vdash loans(s_0'(bb(\sigma, s_0, b_0))) = loans(s_0'(\sigma)) \cup \{b_0\} \\ &\mathbf{pre} \vdash borrower(b_0'(bb(\sigma, s_0, b_0))) = s_0'(\sigma) \end{aligned}$$

where s_0', b_0' are observers on Σ for obtaining the objects s_0, b_0 and $\sigma \in \Sigma$. Notice that, the purpose of precondition **pre** is to guarantee the multiplicity constraints in the class diagram.

In addition to the system operations, an actor may also perform other actions to change the system state. So use cases are more generically specified by integrating the system actions and actor actions together. How can this be specified in our framework? The component calculus mentioned in section 2 provides a set of combinators (notably, for sequential and parallel composition) which can be used for getting more complex use-cases from composing atomic ones. For example, the use case **BB** is defined as a sequential composition of the atomic actions corresponding to events 1 and 2, another use case for returning a book, corresponding to event 3, when the condition is $true$, and event 4.

Use Case	Borrow Book
Actors	Reader, Librarian
Precondition	The book can be borrowed.
Flow of Events	1. The use case begins when the reader chooses a book that is not already lent;
	2. The librarian checks whether the reader could borrow any more books;
	3. If the number of books borrowed by the reader arrived the upper bound Include Return Book;
	4. The librarian assigns the reader as the borrower of the book and states a deadline for returning the book.
Postcondition	The reader has successfully borrowed the book.

Fig. 2. Borrow Book use case

3.4 Statechart Diagrams

In UML *statechart diagrams* describe the dynamic behavior of systems. Several formal semantics for statechart diagrams have been proposed previously. For example, in [13] input-output labelled transition systems are used as the semantic domain. Our aim is to give a coalgebraic characterization of statecharts, so that consistency checks between class diagrams and statechart diagrams became proofs that the coalgebra (statechart) is a model of the corresponding coalgebraic specification (class diagram). Similarly, refinement (implementation) relations can be defined ranging over different view models.

Because of the hierarchical structure of statecharts, we will endow the set of configurations with a coalgebraic structure induced by the operational semantics rather than simply construct the coalgebraic structure over the set of states. Functor T captures the shape of such a coalgebra:

$$\mathsf{T}(X) = \mathsf{B}(X \times \mathcal{P}E)^E$$

where E denotes the set of all events — notice that events, at this stage, may have parameters rather than remaining just primitive signals. B is a strong monad which specifies the behavior pattern of the statechart (typically the powerset monad). For a given statechart SC, let CF be the set of configurations and $(CF, \alpha : CF \to \mathsf{T}(CF))$ the corresponding T-coalgebra. Hence, the behavior of SC is given by

$$\llbracket SC \rrbracket : \lambda s : CF \,.\, \alpha(s)$$

Note that directly defining the coalgebraic semantics may lead to a more faithful model, since we have the freedom to choose the signature functor appropriately. Concerning verification, the coalgebraic approach often allows for a smaller state space, when compared with direct encoding as a Kripke structure, and makes the verification of system properties more effective (as shown by D. Pattinson in [18]). A detailed discussion can be found in [16]. Furthermore, the generic notion of component refinement, proposed by the authors in [15], can be adapted to the context of UML and used for checking consistency of different models. Such is purpose of the following section.

4 Refinement

To make proper use of UML models in a development process we need a clear notion of refinement between components clarifying what it means for a component to implement another. Orthogonal to the *horizontal* decomposition of the system structure, the *vertical* refinement of concrete components from abstract ones provides an approach for stepwise separate development of component-based systems. In the coalgebraic framework sketched in this paper, three kinds of refinement relations between components can be defined. In any case the semantic mapping defined for UML models makes them associated with a proper refinement ordering.

- *Behavioral refinement*, which typically relates components of the same interface, where the refinement is based on a simulation preorder between the two components. Since morphisms between components of the same interface are in fact coalgebra homomorphisms which, therefore, entail bisimilarity, we must seek for a weaker notion of a morphism between components, still preserving the source component dynamics. Concerning UML, this kind of refinement is mainly used for refinement of behavioral models, especially, for statechart diagrams.
- *Interface refinement*, which is concerned with what one may call *plugging compatibility*. It relates components of different interfaces, and the question is whether a component can be transformed, by suitable wiring, to replace another component with a different interface. As the structure of components interface types encodes the available operations, this may capture situations of *extension of component's functionality*, in the sense that the 'concrete' component may introduce new operations, and is used for development with conceptual models, i.e., class diagrams and use cases.
- and *architectural refinement*, which is used for decomposing a component with a specified behavior into a distributed system architecture, i.e., a family of components combined in parallel, which is also modelled as a concrete coalgebra. Such a notion of refinement is mainly used in the development of system architectures, i.e., for component and deployment diagrams.

We say that a component p behaviorally refines component q if the behavioral patterns observed for p are a structural restriction, with respect to the behavioral model captured by monad B, of those of q. To make such a 'definition' more precise we describe behavioral patterns concretely as *generalized transitions*. Thus a possible (and intuitive) way of regarding component p as a behavioral refinement of q is to consider that p transitions are *preserved* in q. For non deterministic components this is understood simply as set inclusion. But one may also want to consider additional restrictions. For example, to stipulate that if p has no transitions from a given state, q should also have no transitions from the corresponding state(s). Recall that a component morphism from p to q is a seed preserving function $h : U_p \longrightarrow U_q$ such that $\mathsf{B}(h \times \mathsf{id}) \cdot \alpha_p = \alpha_q \cdot (h \times \mathsf{id})$. In terms of transitions, this equation is translated into the following two requirements (by a straightforward generalization of an argument in [21]):

$$u \xrightarrow{\langle i,o \rangle}_p u' \;\Rightarrow\; h\,u \xrightarrow{\langle i,o \rangle}_q h\,u' \tag{3}$$

$$h\,u \xrightarrow{\langle i,o \rangle}_q v' \;\Rightarrow\; \exists_{u' \in U}\, s.t.\, u \xrightarrow{\langle i,o \rangle}_p u' \wedge u' = h\,v' \tag{4}$$

which capture the fact that, not only p dynamics, as represented by the induced transition relation, is *preserved* by h (3), but also q dynamics is *reflected* back over the same h (4).

To define a weaker notion of coalgebra morphism, let $\leq$ be an order on a **Set** endo-functor T [10] (concretely, mapping every set U into a collection of preorders $\leq_{\mathsf{T}U}$), referred to as a *refinement preorder*. Then,

Definition 4.1 Let T be an extended polynomial functor on **Set** and consider two T-coalgebras $p = (U, \alpha : U \to \mathsf{T}(U))$ and $q = (V, \beta : V \to \mathsf{T}(V))$. A *forward* morphism $h : p \to q$ with respect to a refinement preorder $\leq$, is a function from U to V such that

$$\mathsf{T}\,h \cdot \alpha \leq \beta \cdot h$$

The existence of a *forward* morphism connecting two components p and q witnesses a refinement situation whose symmetric closure coincides, as expected, with bisimulation. In the sequel we define *behavioral refinement* as the existence of a forward morphism *up to bisimulation* [5]. Formally,

Definition 4.2 Given components p and q, p is a *behavioral refinement* of q,

[5] In [15] the dual notion of a *backwards* morphism, *i.e.*, one that satisfies $\beta \cdot h \leq \mathsf{T}\,h \cdot \alpha$, is also studied, leading to a notion of *backward* refinement which do have some applications, although the underlying intuition seems less familiar.

written $p \sqsubseteq_B q$, if there exist components r and s such that $p \sim r$, $q \sim s$ and there exists a (seed preserving) forward morphism from r to s.

Behavioral refinement characterizes the preservation of component behavior. But if we rely solely on behavioral refinement, the inability to change the syntactic interface will force us to work at the same level of interface abstraction throughout the whole development process. To avoid this, a more general notion of refinement, called interface refinement is introduced, which relates components with different interfaces.

Definition 4.3 Let $p : I \to O$ and $q : I' \to O'$ be components. If there exist functions $w_1 : I' \to I$ and $w_2 : O \to O'$, such that

$$p[w_1, w_2] \sqsubseteq_B q$$

then p is an *interface refinement* of q modulo the downwards function w_1 and the upwards function w_2, written as $p \sqsubseteq_{(w_1,w_2)} q$.

Interface refinement supports the systematic construction of new components from existing ones. Generally, for any component p, and functions w_1, w_2,

$$p \sqsubseteq_{(w_1,w_2)} p[w_1, w_2]$$

One situation where this technique is useful is when we have an already completed off-the-shelf component and want to adapt the syntactic interface of this component to fit some context requirements. Therefore, interface refinement provides a systematic pattern for interface adaptation of components.

Both kinds of refinement relations introduced above are defined with respect to the underlying black-box behavior of components. However, when we develop a system, it is certainly not enough to characterize its black-box behavior only, we also need to capture some internal structural aspects. For this purpose, we introduce the notion of *architectural refinement.*

For this let us define an *architecture* as a tuple $S = \langle I, O, C, R\rangle$ where I and O are the input and output interface of the system, respectively, $C = \{p_k = \langle U_k, \overline{\alpha}_k, u_k\rangle\}_{k=1,2,\cdots,n}$ denotes a finite set of components, and R denotes a finite set of combinators together with the components being combined by them. In fact, systems can be decomposed hierarchically, and regarded as ordinary components again, which means that the notion of behavioral refinement remains applicable. The architectural refinement relation is then therefore defined as a behavioral refinement within the given interface:

Definition 4.4 Let S and S' be two systems. If $I = I'$, $O = O'$, and $[\![S]\!] \sqsubseteq_B [\![S']\!]$, then we say that S is an architectural refinement of S', written as $S \sqsubseteq_A S'$.

The most obvious use of a notion of refinement is to compare two alternative designs for the same component. We take every design as a separate model in UML and compare them at the semantic level by defining their semantics as coalgebras. For M_1 and M_2 UML models, we define

$$\begin{array}{c} M_1 \sqsubseteq_B M_2 \Leftrightarrow [\![M_1]\!] \sqsubseteq_B [\![M_2]\!] \\ M_1 \sqsubseteq_{(w_1,w_2)} M_2 \Leftrightarrow [\![M_1]\!] \sqsubseteq_{(w_1,w_2)} [\![M_2]\!] \\ M_1 \sqsubseteq_A M_2 \Leftrightarrow [\![M_1]\!] \sqsubseteq_A [\![M_2]\!] \end{array}$$

Another use of the refinement order is to check the consistency between models of the same system.

5 Conclusions and Future Work

We have presented a generic semantic framework for component-based development starting from UML descriptions, extending previous work on componentware detailed on a series of papers under the slogan “components as coalgebras” [1,2,3,15], where components are made generic in the sense that their behavioral patterns are described by strong monads acting as parameters in the calculus. Essential parts of a coalgebraic semantics for UML class diagrams have been presented in this paper. It has been shown in detail how classes, associations and generalizations in a UML diagram can be interpreted as coalgebraic specifications. Furthermore an outlook on the formalization of object, use-case and statechart diagrams has also been given. Although, space limitation preclude a more technical discussion of other diagrams, the reader should by now get an idea how a coalgebraic semantics facilitate the integration of static and dynamic aspects of UML.

Another major aspect of our framework is refinement, which formalizes the replaceability of components. Refinement is a very basic idea in sequential programming whose foundations can be traced back to Hoare’s landmark paper [6] (see [5] for a recent account). However, for component-based systems, the situation is far more complex (see, *e.g.*, [4]). In this paper we investigated refinement at three, inter-related, levels: behavioral, interface and architectural level. Our framework allows the notion of refinement to change the interface and granularity of components.

A major influence in our work was Jacobs and Tews research on object-oriented systems [8,9,23]. However, instead of defining and resorting to a specific (coalgebraic) specification language (CCSL), we adopt a rather pragmatic approach: that of using UML diagrams to denote coalgebraic specifications. Another difference between our work and the formalization of UML in [23] is that we do not restrict ourselves to class diagrams.

Actually, this paper intends to be a step towards a unifying coalgebraic semantics for UML. Currently, our research programme includes further work on use cases, statechart diagrams, and suitable notions of consistency among UML models. Further UML elements, such as interaction diagrams, will be addressed at a later stage. The calculus provided in the framework is a promising candidate to model the interactions. However, having a formal semantics is not enough. The next step will involve research on applications of the semantics: our long term research aim is to propose a use-case driven development method for UML designs, supported by algebraic laws and coinductive proof methods. On the technological side, the approach will be supported by a combination of a model checker and a test-case generator for UML diagrams.

References

[1] Luís S. Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.

[2] Luís S. Barbosa and Josè Nuno Fonseca de Oliveira. State-based components made generic. In H. Peter Gumm, editor, *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, volume 82.1, Warsaw, April 2003.

[3] Luís S. Barbosa and Sun Meng. Generic components. In Graham Hutton, editor, *Proceedings of First APPSEM-II Workshop*, Nottingham, March 2003. APPSEM Network Report.

[4] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems : focus on streams, interfaces, and refinement*, volume 62 of *Monographs in computer science*. Springer, 2001.

[5] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[6] Charles Antony Richard Hoare. Proof of correctness of data representations. *Acta Information*, 1:271–281, 1972.

[7] Bart Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, volume 1098 of *LNCS*, pages 210–231. Springer, Berlin, 1996.

[8] Bart Jacobs. Objects and classes, co-algebraically. In B. Freitag, C. Lengauer C.B. Jones, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer, 1996.

[9] Bart Jacobs. Exercises in coalgebraic specification. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, pages 237–280. Springer, 2002.

[10] Bart Jacobs and Jesse Hughes. Simulations in coalgebra. In H. Peter Gumm, editor, *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, volume 82, pages 245–263, Warsaw, April 2003.

[11] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[12] Ivar Jacobson. Object Oriented Development in an Industrial Environment. In Norman K. Meyrowitz, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), October 4-8, 1987, Orlando, Florida, Proceedings*, volume 22 of *SIGPLAN Notices*, pages 183–191, 1987.

[13] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99*. Kluwer, 1999.

[14] Sun Meng and Bernhard Aichernig. Towards a Coalgebraic Semantics of UML: Class Diagrams and Use Cases. Technical Report 272, UNU/IIST, January 2003.

[15] Sun Meng and Luís S. Barbosa. On Refinement of Generic State-based Software Components. In *Proceedings of 10th International Conference on Algebraic Methods And Software Technology, AMAST'04*, LNCS. Springer, 2004.

[16] Sun Meng, Zhang Naixiao, and Luís S. Barbosa. On Semantics and Refinement of Statecharts: A Coalgebraic View. In *Proceedings of 2nd IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society Press, 2004.

[17] OMG. *OMG Unified Modeling Language Specification, Version 1.4* , 2001.

[18] Dirk Pattinson. Coalgebraic techniques in modelchecking, 2001. Available from http://siskin.pst.informatik.uni-muenchen.de/ pattinso/Publications/.

[19] Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.

[20] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, 1999.

[21] Jan Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.

[22] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[23] Hendrik Tews. *Coalgebraic Methods for Object-Oriented Specification*. PhD thesis, TU Dresden, 2002.