Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 135 (2006) 73–84
www.elsevier.com/locate/entcs

Coalgebraic Description of Generalized Binary Methods[1]

Furio Honsell[2]

DIMI, Università di Udine, ITALY

Marina Lenisa[3]

DIMI, Università di Udine, ITALY

Rekha Redamalla[4]

DIMI, Università di Udine, ITALY, and
B.M. Birla Science Centre, Hyderabad, INDIA.

Abstract

We extend the Reichel-Jacobs coalgebraic account of specification and refinement of objects and classes in Object Oriented Programming to (*generalized*) *binary methods*. These are methods that take more than one parameter of a class type. Class types include sums and (possibly infinite) products type constructors. We study and compare two solutions for modeling generalized binary methods, which use purely covariant functors. In the first solution, which applies when we already have a class implementation, we reduce the behaviour of a generalized binary method to that of a bunch of unary methods. These are obtained by *freezing* the types of the extra class parameters to constant types. The *bisimulation behavioural equivalence* induced on objects by this model amounts to the *greatest congruence* w.r.t method application. Alternatively, we treat binary methods as *graphs* instead of functions, thus turning contravariant occurrences in the functor into covariant ones.

Keywords: OO-programming, Binary methods, Coalgebraic semantics.

[1] Work partially supported by the UE Project IST-510996 TYPES.
[2] Email: honsell@dimi.uniud.it
[3] Email: lenisa@dimi.uniud.it
[4] Email: redamall@dimi.uniud.it

doi:10.1016/j.entcs.2005.09.022

Introduction

In [10,7,8], a categorical semantics for *objects* and *classes* based on *coalgebras* is given. The idea underpinning this approach is that coalgebras, duals of algebras, allow to focus on the behaviour of objects while abstracting from the concrete representation of the state of the objects.

In the coalgebraic approach of [10,7,8], a class is modelled as an F-coalgebra $(A, f : A \to F(A))$ for a suitable functor F. The carrier A represents the space of *attributes*, or *fields*, and the coalgebra operation f represents the *public* methods of the class, *i.e.* the methods which are accessible from outside the class. Thus the objects of a class are modelled as the elements of the carrier. Their behaviour under application of public methods, viewed as functions acting on objects, is then captured by the coalgebra map f. Thus the coalgebraic model induces exactly the behavioural equivalence on objects, whereby two objects are equated if, for each public method, the application of the method to the two objects, for any list of parameters, produces equivalent results. A benefit of the coalgebraic model is a coinduction principle for establishing the behavioural equivalence.

Following [7], we distinguish between class *specifications* and class *implementations* (or simply classes). A class specification is like an abstract class, in which only the signatures of constructors and (public) methods are given, without their actual code. Assertions enforce behavioural constraints on constructors and methods. Implementation of constructors and methods is given in a class implementation. In the *bialgebraic* approach, a class specification induces a pair of functors, determined by the signature of constructors and methods, respectively. A class implementation is any bialgebra satisfying the assertions. Here we will focus only on the coalgebraic part, which is the problematic one. For a complete bialgebraic treatment, see [6].

Binary methods, *i.e.* methods with more than one class argument, apparently escape the coalgebraic approach. The extra class parameters produce contravariant occurrences in the functor modelling methods, and hence cannot be dealt with a straightforward application of the coalgebraic methodology.

We extend the Reichel-Jacobs coalgebraic description to generalized binary methods, *i.e.* methods whose type parameters include sums and possibly infinite products type constructors. Our focus of interest are equivalences on objects which are "well-behaved", in the sense that they induce a canonical non-redundant model on the quotient of the given class. Therefore, such equivalences must be *congruences* w.r.t. method application. In this paper we show that canonical models can be built also for generalized binary methods using purely covariant tools. We propose two solutions. Our first solution

applies to the case where we have already a class implementation. It is based on the observation that the behaviour of a generalized binary method can be captured by a bunch of unary methods obtained by *"freezing"*, in turn, the types of the class parameters to the states of the class implementation given at the outset, *i.e.* by viewing them as constant types. Our second solution is based on a set-theoretic understanding of functions, whereby binary methods in a class specification are viewed as *graphs* instead of functions. Thus contravariant function spaces in the functor are rendered as covariant sets of relations.

We prove that the behavioural equivalence induced by the *"freezing approach"* amounts to the *greatest congruence* w.r.t method application on the given class. As a by-product, we gain a (coalgebraic) coinduction principle for reasoning about such greatest congruence.

As far as the graph model is concern, the behavioural equivalence is not a congruence, in general. Remarkably, we show that a necessary and sufficient condition for this to hold is that the graph and freezing equivalence coincide. As a consequence, when this is the case, we obtain a spectrum of coinduction principles for reasoning on the greatest congruence.

The interest of the graph approach goes beyond coalgebraic semantics, since it suggests a new way for solving the well-known problem of typing binary methods when subclasses are viewed as subtypes, *e.g* see [3].

In this paper, we work on a set-theoretic category, denoted by $\mathcal{C}$. For basic definitions and results on coalgebras we refer to [9].

In the literature, various authors have been considered the problem of the coalgebraic description of binary methods, *e.g* see [12]. For an extensive comparison with the literature, see [6].

1 Generalized Binary Methods and Behavioural Equivalences

We call a method $m : X \times T_1 \times \ldots \times T_q \to T_0$ *generalized binary* if T_i ranges over the following grammar of types:

$$(\mathcal{T} \ni)\ T ::= \ X \mid K \mid T \times T \mid T + T \mid \Pi_K T,$$

where $X \in TVar$, is a variable for class types, and K is any constant type. Notice that the product type $\Pi_K T$ corresponds to the function space $K \to T$. That is, in a generalized binary method, we allow functional parameters, where variable types can appear only in *strictly positive* positions. For simplicity, in this paper we will consider only one class. There would be no additional

conceptual difficulty in dealing with the general case.

A preliminary step in discussing equivalences induced on objects by generalized binary methods consists in extending the behavioural equivalence on objects of a class X to the whole structure of types in $\mathcal{T}$ over X. This is achieved through the *relational lifting* of [5]. In the definition below, by abuse of notation, we denote by X and T also their set-theoretic semantic counterparts.

Definition 1.1 [Relational Lifting] Let R^X be a relation on X, let $T \in \mathcal{T}$ be such that $Var(T) \subseteq \{X\}$. We define the extension $R^T \subseteq T \times T$ by induction on T as follows:

- if $T = K$, then $R^T = Id_{K \times K}$,
- if $T = T_1 \times T_2$, then $R^T = \{((a_1, a_2), (a_1', a_2')) \mid a_1 R^{T_1} a_1' \wedge a_2 R^{T_2} a_2'\}$,
- if $T = T_1 + T_2$, then $R^T = \{((1, a), (1, a')) \mid a R^{T_1} a'\} \cup \{((2, a), (2, a')) \mid a R^{T_2} a'\}$,
- if $T = \Pi_K T_1$, then $R^T = \{(f, f') \in \Pi_K T_1 \mid \forall a \in K \implies f a R^{T_1} f' a\}$.

Definition 1.2 [Congruence] Let $\approx^X$ be an equivalence on objects of a class X and let $m : X \times T_1 \times \ldots \times T_q \to T_0$ be a method in X, then $\approx^X$ is a *congruence w.r.t. m* if $x \approx^X x'$ and $a_1 \approx^{T_1} a_1' \ldots a_q \approx^{T_q} a_q' \Rightarrow x.m(\boldsymbol{a}) \approx^{T_0} x'.m(\boldsymbol{a'})$, where $\approx^{T_i}$ denotes the extension of $\approx^X$ to the type T_i, according to the definition above. (Notice the use of the "dot-notation" for method calls.)

2 Class Specifications and Class Implementations

Definition 2.1 A class specification S is a structure consisting of

- A finite set of *method declarations*

$$m : X \times T_1 \times \ldots \times T_q \to T_0 \ ,$$

- A finite set of *assertions*, regulating the behaviour of the objects belonging to the class.

The language for assertions is any first order language with constant symbols and function symbols for denoting constructors, methods and (extensions of) behavioural equivalences with all types. Typical assertions are equations, *e.g* see [11] for more details.

In Table 1, on the lefthand part, we present an example of a class specification, *Register*, which features binary method *eq* for comparing the content of two registers.

A class (implementation) consists of attributes (fields), constructors and methods. Attributes and methods of a class can be either private or public.

class spec : *Register*	**class** R
methods :	**attributes** :
set : $X \times N \to X$	$val : int$
get : $X \to N$	**methods** :
eq : $X \times X \to \mathbb{B}$	r.get = r.val
assertions :	r.set(n) = r′
r.set(n).get = n	where r′.$val = n$
r_1.get = r_2.get ⇔	r_1.eq(r_2) = **if** (r_1.get = r_2.get)
r_1.eq(r_2) = $true$	**then** $true$
end class spec	**else** $false$
	end class

Table 1
Example of Class Specification and Class.

For simplicity, we assume all attributes to be private, and all methods to be public. We do not use a specific programming language to define classes, since we are working at a semantic level. Any programming language would do. In this perspective, the code corresponding to a method declaration $m : X \times \prod_{j=1}^{q} T_j \to T_0$ is given by set-theoretic function $\alpha : X \times \prod_j^q T_j \to T_0 + Excp + 1$, since method can possibly terminate with an exception or not terminate.

Definition 2.2 A class C *implements* a specification S if method declarations correspond, and their implementations satisfy the assertions in S.

In Table 1, on the righthand part, we present the class R implementing the class specification, *Register*.

3 Coalgebraic Description of Objects and Classes: unary case

In this section, we illustrate the coalgebraic description of class specifications and class implementations in the case of unary methods. Following [10,7], we associate a functor to a class specification as follows:

Definition 3.1 Let S be a class specification with method declarations $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}, \quad i = 1, \ldots, k$, where all methods are unary (*i.e.* $X \notin T_{ij} \ \forall j = 1, \ldots, q_i$). Then using currification the method declarations in S induce the functor $H : \mathcal{C} \to \mathcal{C}$ defined by

$$H \triangleq \prod_{i=1}^{k} \prod_{j=1}^{q_i} T_{ij} \to (T_{i0} + Excp + 1),$$

where $Excp$ denotes a set of exceptions/errors.

Notice that the functor H is covariant only if the method m is unary. Generalized binary methods, such as the method *eq* in the class specification *Register*, produce contravariant occurrences of X in the corresponding functor. In Section 4, we discuss how to overcome this problem.

The class implementations can be viewed as coalgebras as follows:

Definition 3.2 Let S be a class specification inducing a functor H. A *class* implementing S is an H-coalgebra satisfying the assertions in S.

On the other hand, given a concrete class, this induces a coalgebra for the functor determined by its method declarations, as follows:

Definition 3.3 i) A class $C = \langle \{f_i : T_i\}_{i=1}^n, \{m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}\}_{i=1}^k \rangle$ induces a coalgebra (X, α) for the functor H determined by the declarations of methods m_i, defined as follows:

- The carrier X is the set of *states* determined by the fields f_i.
- The coalgebra map $\alpha : X \to HX$ is defined by $\alpha \triangleq \langle \alpha_i \rangle_{i=1}^k$, where $\alpha_i : X \to H_i X$ is the function implementing the method m_i.

ii) An object of a class C is an element of the set of states X of C.

In the following lemma we characterize the behavioural equivalence on objects induced by the coalgebraic description of a class implementation. Such behavioural equivalence equates objects with the same behaviour under application of methods:

Lemma 3.4 *Let S be a class specification with method declarations $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i_0}$, $i = 1, \ldots, k$, inducing the functor $H = \prod_{i=1}^k H_i$, let $(X, \langle \alpha_i \rangle_{i=1}^k)$ be an H-coalgebra implementing S. Then the greatest H-bisimulation on $(X, \langle \alpha_i \rangle_i)$, $\approx_H$, can be characterized as follows:*

$$x \approx_H x' \iff \forall i.\ \forall \boldsymbol{a}.\ \alpha_i(x)(\boldsymbol{a}) \approx_H \alpha_i(x')(\boldsymbol{a}) \ ,$$

where, by abuse of notation, $\alpha_i(x)(\boldsymbol{a}) \approx_H \alpha_i(x')(\boldsymbol{a})$ denotes the extension of $\approx_H$ to the type T_{i0}, i.e. $\approx_H^{T_{i0}}$, *according to Definition 1.1 of relational lifting.*

4 Coalgebraic Description of Generalized Binary Methods

In this section, we show how to extend the coalgebraic model to generalized binary methods. Our first proposal (Section 4.1) applies when a concrete

coalgebra (*i.e.* class implementation) is given. It is based on the observation that the behaviour of a generalized binary method can be simulated by a bunch of unary methods, each one determined by "freezing" all the occurrences of X in the parameter types and object type, but one. "Freezing" an occurrence of X means that X is replaced by the carrier, *i.e.* the set of states, of the given class. The behavioural equivalence thus obtained turns out to be the greatest congruence w.r.t. the original generalized binary method.

In Section 4.2, we present an alternative solution to the freezing functor. Here we turn contravariant occurrences in the type of parameters of a generalized binary method m into covariant ones simply by interpreting m as a *graph* instead of a function. To this aim, we introduce a new functor G (*graph functor*), where the function space is substituted by the corresponding space of *graph relations*.

The advantage of this latter solution w.r.t. the previous one is that this approach directly applies to specifications. Moreover, we do not have to use the intermediate step of the unary methods. The drawback is that the graph behavioural equivalence is not a congruence w.r.t. method application in general. However, there are many interesting situations where it is. In these cases a rich spectrum of conceptually independent coinduction principles is available. We discuss this issue in Section 4.3.

4.1 The Freezing Functor

We proceed in two passes. First, we reduce a generalized binary method to a bunch of purely binary methods with the same observable behaviour. Let m be a generalized binary method. In particular, for each parameter of type $T_1 + T_2$, we can duplicate the method. In the first version, we will have a parameter of type T_1. In the second version the parameter will be of type T_2. Moreover, each parameter of type $\Pi_K T$ can be viewed as the product of $|K|$ parameters of type T. Thus, by applying the above transformations to a generalized binary method, we get a (possibly infinite) set of purely binary methods $m : X \times \prod_{j\in J} T_j \to T_0$, where J is a possibly infinite set of indexes.

In the second pass, we reduce each purely binary method to a bunch of unary methods. Let C be a class implementation with set of states $\bar{X}$, including a purely binary method $m : X \times \prod_{j\in J} T_j \to T_0$, implemented by the function α. In order to recover the observable behaviour of the original method m, we need to consider a bunch of unary methods m_l, one for each class parameter, where m_l describes the behaviour of an object when it is used as l^{th} class parameter. Let I be the set of indexes corresponding to class

parameters, including the object, for all $l \in I$, we define :

$$m_l : X \times (\prod_{j\in J} T_j[\bar{X}/X]) \to T_0, \quad \alpha_l : X \times (\prod_{j\in J} T_j[\bar{X}/X] \to T_0)$$

$$\alpha_l(x)(a_1, \ldots, a_q) \triangleq \alpha(a_l)(a_1, \ldots, a_{l-1}, x, a_{l+1}, \ldots, a_q) \ .$$

Now we can define a coalgebraic model of the class implementation using purely covariant tools, as in Section 3, using the freezing functor F defined by:

Definition 4.1 [Freezing Functor] Let C be a class implementation with purely binary methods and set of states $\bar{X}$. The freezing functor determined by C is defined by

$$F \triangleq \prod_{i=1}^{k} F_i \ ,$$

where, for each unary method $m_i, F_i \triangleq H_i$, and for each binary method $m_i : X \times \prod_{j\in J_i} T_{ij} \to T_{i0}$ with class parameters in I_i, $F_i \triangleq \prod_{l\in I_i} F_{il}$, where $F_{il}X \triangleq (\prod_{j\in J} T_{ij})[\bar{X}/X] \to (T_{i0} + Excp + 1)$, for all $l_i \in I_i$.

The following definition of *1-ary method context* will be useful to characterize the behavioural equivalence induced by the freezing model:

Definition 4.2 [1-ary Method Context] Let C be a class. *A 1-ary method context*, $D[\ ]$, is a context with exactly one hole, whose top operator is a method in C, where the hole either corresponds to the object or to a parameter

The behavioural equivalence on objects induced by the freezing functor can then be characterized as follows:

Lemma 4.3 (Freezing Bisimulation and Coinduction Principle) *Let C be a class implementation with set of states $\bar{X}$, and let F be the freezing functor induced by C. Then the greatest F-bisimulation $\approx_F$, on the coalgebra determined by the methods in C, can be characterized as follows:*

$$x \approx_F x' \iff \forall D[\ ] \text{ 1-ary context. } D[x] \approx_F D[x']$$

We can now establish the result which motivates our treatment:

Theorem 4.4 *Let C be a class. Then for all methods $m_i : X \times \prod_{j\in J} T_{ij} \to T_{i0}$ in $C, \forall x\boldsymbol{a}, x'\boldsymbol{a'} \in X \times \prod_{j\in J} T_{ij}$, $x\boldsymbol{a} \approx_F x\boldsymbol{a'} \implies \alpha_i(x)(\boldsymbol{a}) \approx_F \alpha_i(x')(\boldsymbol{a'})$.*

Moreover, since any congruence is an F-bisimulation, by coinduction, one can then show that $\approx_F$ is the greatest congruence.

Theorem 4.5 *Let C be a class. Then the freezing behavioural equivalence $\approx_F$ induced on C is the greatest congruence w.r.t method application.*

4.2 The Graph Functor

Definition 4.6 [Graph Functor] The method declarations in S induce the *graph functor* $G : \mathcal{C} \to \mathcal{C}$ defined by

$$G \triangleq \prod_{i=1}^{k} G_i \ ,$$

where, for each unary method $m_i, G_i \triangleq H_i$ (see Definition 3.1), and for each generalized binary method $m_i : X \times \prod_{j\in J} T_{ij} \to T_{i0}$, $G_i \triangleq \mathcal{P}(\prod_{j\in J} T_{ij} \times (T_{i0} + Excp + 1))$.

Definition 3.3, which gives the coalgebra induced by a given class, extends immediately to the case of the graph functor. On the contrary, the extension to the graph functor of the definition of class implementation (Definition 3.2) requires more care. Namely class implementations shall be taken to be *functional* G-coalgebras.

The graph behavioural equivalence can be characterized in terms of *n-ary method contexts*, which are method contexts with holes for any class parameter.

Definition 4.7 [n-ary Method Context] Let C be a class, and let m be a method of C with n (generalized) class parameters including the object, implemented by α. The method m induces an *n-ary context*

$$D[\ ] = [\ ].\alpha(b_1, \ldots, b_k) \ ,$$

where $b_i = [\ ]$, if T_i is a (generalized) class type, otherwise, if T_i is a constant type, b_i is any argument of type T_i.

Lemma 4.8 (Graph Bisimulation and Coinduction Principle) *Let $G = \prod_{i=1}^{k} G_i$ be the functor induced by the method declarations in S, and let $(X, \langle\alpha_i\rangle_{i=1}^{k})$ be a G-coalgebra implementing S. Then the greatest G-bisimulation on $(X, \langle\alpha_i\rangle_{i=1}^{k})$, $\approx_G$, can be characterized as follows:*

$$x \approx_G x' \iff \forall D[\ ] \textit{n-ary context.}\ \ \forall \boldsymbol{a}\exists \boldsymbol{a'}.(\boldsymbol{a} \approx_G \boldsymbol{a'} \ \&\ D[x,\boldsymbol{a}] \approx_G D[x,\boldsymbol{a'}])$$

$$\&\ \forall \boldsymbol{a'}\exists \boldsymbol{a}.(\boldsymbol{a} \approx_G \boldsymbol{a'} \ \&\ D[x,\boldsymbol{a}] \approx_G D[x,\boldsymbol{a'}]) \ .$$

Notice the alternation of quantifiers $\forall\exists$ in the definition of graph behavioural equivalence, due to the presence of the powerset in the graph functor.

The functor G has always a final coalgebra, *e.g.* see [1]. In general, it is not functional, and moreover the functionality property of a coalgebra is not preserved by the unique morphism into the final coalgebra. Therefore, the image of a class implementation under the final morphism is not guaranteed to be a class implementation. Thus we can lack minimal class implementations. In Section 4.3, we study conditions for the final morphism to preserve the functionality property, thus recovering minimal implementations.

4.3 Comparing Graph and Freezing Behavioural Equivalences

One can easily check that $\approx_F$ is a graph bisimulation, using reflexivity of $\approx_F$. Thus $\approx_F \subseteq \approx_G$. The converse inclusion does not hold in general. For example, this is the case for the class R' obtained from the class R of registers when we drop methods *get* and *set*, and we consider only method *eq*. Namely, for R', $\approx_G$ equates all pairs of registers, while $\approx_F$ is the identity relation on registers. Moreover, notice that in this case $\approx_G$ is *not* a congruence w.r.t. *eq*.

The following result is a fundamental tool for recovering $\approx_F = \approx_G$:

Theorem 4.9 *$\approx_G = \approx_F$ iff $\approx_G$ is a congruence w.r.t. the methods in the class.*

The equality $\approx_G = \approx_F$ on a functional G-coalgebra is equivalent to the fact that its image into the final coalgebra is still a functional coalgebra. Thus Theorem 4.9 above gives an answer to the problem of minimal class implementations for the graph functor, raised at the end of Section 4.2.

Another relevant consequence of the fact that $\approx_G = \approx_F$ is a simplified coinductive characterization of $\approx_F$, in terms of "head" contexts, where the hole is in head position, *i.e.* it corresponds to the target object:

Proposition 4.10 *If $\approx_G = \approx_F$, then*

$$x \approx_F x' \iff \forall D[\ ] \text{ head context. } D[x] \approx_F D[x'].$$

Theorem 4.9 above is all that we might want. However, in practice, it is useful to have also alternative sufficient conditions. The interested reader can see [6].

5 Relational types

The idea of treating binary methods as graphs, rather than as functions, can be fruitfully pursued to overcome the well-known problem arising when in-

$$\frac{x:\alpha \quad M:\beta}{\lambda x^{\alpha}.M:\alpha\otimes\beta} \qquad \frac{\alpha\leq\alpha' \quad \beta\leq\beta'}{\alpha\otimes\beta\leq\alpha'\otimes\beta'} \qquad \frac{}{\alpha\to\beta\leq\alpha\otimes\beta}$$

Table 2
Typing rules for *Relational Types* $\otimes$

heritance is combined with subtyping, *e.g.* see [3]. Namely, if we type binary methods with the usual arrow type, which is contravariant, we lose the property that subclasses are subtypes. We propose to introduce a new type constructor, *i.e.* the *relation type*, and use this to type binary methods in class declarations. Since relation types are purely covariant, the subtyping property is maintained by subclasses. Binary methods can still be typed also with the standard arrow type, which is a subtype of the corresponding relation type, see Table 2. To preserve safety, contrary to arrow types, we assume relation types not to be "applicable" *i.e.* there is no relational counterpart to the rule : $\frac{M:\alpha\to\beta \quad N:\alpha}{MN:\beta}$. This solution to the problem of typing binary methods is quite simple, and it allows for *single dispatching* in method calls. Moreover, contrary to other proposals, our proposal allows for "future code extensions" without losing the subtyping property of classes. We will study this proposal in a future paper.

References

[1] P.Aczel. *Non-wellfounded sets*, CSLI Lecture Notes **14**, Stanford 1988.

[2] Aczel P., N.Mendler. A Final Coalgebra Theorem, *CTCS*, D.H.Pitt et al. eds., Springer LNCS **389**, 1989, 357–365.

[3] Bruce K.B., Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, Benjamin C. Pierce On Binary Methods, *TAPOS* **1**(3), 1995, 221-242.

[4] Forti M., F.Honsell. Set-theory with free construction principles, *Ann. Scuola Norm. Sup. Pisa*, Cl. Sci. (4)**10**, 1983, 493–522.

[5] Hermida C., B.Jacobs. Structural induction and coinduction in a fibrational setting, Information and Computation, 1998, 145(2):107-152.

[6] Honsell F., M.Lenisa, R.Redamalla. Coalgebraic Description of Generalized Binary Methods, TR 8/2005, University of Udine(Italy), 2005.

[7] Jacobs B.. Objects and Classes, co-algebraically, *Object-Orientation with Parallelism and Book Persistence*, B.Freitag et al. eds., Kluwer Academic Publishers, 1996, 83–103.

[8] Jacobs B. Behaviour-refinement of object-oriented specifications with coinductive correctness proofs, *TAPSOFT'97*, M.Bidoit et. al. eds., Springer LNCS **1214**, 1997, 787–802.

[9] Jacobs B., J.Rutten. A tutorial on (co)algebras and (co)induction, *Bulletin of the EATCS* **62**, 1996, 222–259.

[10] Reichel H. An approach to object semantics based on terminal co-algebras, *MSCS* **5**, 1995, 129-152.

[11] Rothe J., H. Tews and B. Jacobs. The Coalgebraic Class Specification Language CCSL, Journal of Universal Computer Science,**7**(2001), pp.175-193.

[12] Tews H. Coalgebraic Methods for Object-Oriented Specifications, Ph.D. thesis, Dresden Univ. of technology, 2002.