UML Model Refactoring as Refinement: A Coalgebraic Perspective

L. S. Barbosa
DI - CCTC
Minho University
Braga, Portugal
Email: lsb@di.uminho.pt

Sun Meng
CWI
Kruislaan 413
Amsterdam, The Netherlands
Email: M.Sun@cwi.nl

***Abstract*—Although increasingly popular, Model Driven Architecture (MDA) still lacks suitable formal foundations on top of which rigorous methodologies for the description, analysis and transformation of models could be built. This paper aims to contribute in this direction: building on previous work by the authors on coalgebraic refinement for software components and architectures, it discusses refactoring of models within a coalgebraic semantic framework. Architectures are defined through aggregation based on a coalgebraic semantics for (subsets of) UML. On the other hand, such aggregations, no matter how large and complex they are, can always be dealt with as coalgebras themselves. This paves the way to a discipline of models' transformations which, being invariant under either behavioural equivalence or refinement, are able to formally capture a large number of refactoring patterns. The main ideas underlying this research are presented through a detailed example in the context of refactoring of UML class diagrams.**

I. Introduction

Model Driven Software Engineering (MDSE) [25] is currently a highly praised development paradigm among software developers and researchers. Its key message: software systems can be developed, enhanced, and maintained through successive refinement and transformation of *models* at various levels of abstraction. Therefore, rather than directly addressing concrete programs or low-level descriptions, the primary artifacts in MDSE are models themselves and transformations. Its intuitions resort to combine architectures of the model roles with other process activities, and to use model rather than code as the driving force of software development.

The Unified Modeling Language (UML) [22] provides a unified notation for representation of various models, and has been widely adopted for the representation of aspects of distributed, web-based applications [7]. According to [22], UML is *'a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system'*. In practice, it stands for a collection of inter-related, semi-formal design notations for software development, providing a unified notation, expressive and widely adopted (a *de facto* standard). It lacks, however, a rigourous and consensual semantic definition leading, therefore, to weak effective support to the design of complex systems and, often, to conflicting support tools.

A variant of MDSE is the Model Driven Architecture (MDA) proposed by the Object Management Group (OMG) [21]. MDA provides an enabling infrastructure with standard specifications facilitating the definition and implementation of model transformations.

An important issue in MDA is *model refactoring*. The term refactoring was originally introduced in [23] in the context of OO programming. In [6], it is defined as the process of *'changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure'*. Later, research has shifted from program refactoring to model refactoring [26], [27], which aims to apply refactoring techniques at the model level instead of program (source code) level.

According to the definition above, refactoring should preserve behaviours when a model is transformed. Unfortunately, a precise definition of behaviour is rarely provided. The original definition of behaviour preservation in [23] states that for the same inputs, the resulting outputs should be the same before and after the refactoring. Such an understanding of behaviour preservation is ensured by means of refactoring preconditions in [23]. However, this is a rather conservative approach which rules out many legal refactorings. Moreover, requiring preservation of I/O behaviour is either insufficient or excessive in many application domains.

A graph transformation approach was proposed by Mens et al. in [19], [18] to provide formal support for refactoring. A type checking approach is used in [28] to ensure that the type of an entity is same before and after refactoring. These approaches are static and mostly operate at source code level. In [29], UML statecharts are translated into CSP processes, and behaviour preservation in refactoring is witnessed by failure-divergence refinement in CSP. Unfortunately, it only deals with the statechart model, while in real application domains, many aspects of behaviour may be relevant. Thus a generic definition of behaviour preservation is still needed for model refactoring. Other interesting approaches include [27] and [11]. Reference [20] provides a comprehensive survey.

In previous work, we introduced a generic coalgebraic semantic framework for different models in UML, including class diagrams, use cases, statecharts and sequence diagrams [12], [17], [15]. In such a framework, the semantics of different kinds of models are given as coalgebras [9], [24] which encapsulate a state space, regarded as a black box with limited access

via specific observers. Notions of bisimulation and refinement capture observational equivalence and simulation preorders, respectively. Such standard tools in coalgebra theory can form the basis of a whole discipline of reasoning and transforming UML designs. Actually, if the semantics of different UML models can be presented as coalgebras for suitable functors, we will end up with a *uniform setting* for tackling the diversity of such models, their properties and inter-relations.

This paper is part of such a broad research agenda on a *generic coalgebraic semantic framework for UML descriptions* upon which we aim at addressing two main issues in model-driven development:

- model *composition*, defining and investigating operators and laws which govern the behaviour of models, and
- model *refactoring*, understood as *the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure*, to quote [23].

In both cases a precise notion of *behaviour* and a calculational approach to *behavioural equivalence* and *refinement* is the key issue. Actually, such notions are at the kernel of coalgebra theory, often suitably called the *mathematics of dynamical systems*. In particular, coalgebra theory provides a standard notion of systems' behaviour in terms of the bisimilarity relation induced by the signature functor. Refinement, as explained below, corresponds to the ability of a coalgebra to simulate another in a quite precise, but parametric, way.

A specific contribution of the paper is the case for taking (coalgebraic) refinement [14], instead of observational equivalence, as the basic notion capturing the intuitive idea of 'behaviour preservation' under refactoring. Often, in fact, equivalence, or in coalgebraic terms, bisimilarity, is not coarse enough to capture typical refactoring patterns. This will be made clear even in the quite elementary example of a refactoring pattern for Class Diagrams used as a case-study in the sequel.

The remaining of the paper is organised as follows: Section II recalls basic concepts in coalgebra theory, with a particular emphasis in our own approach to generic refinement as detailed in [14]. Section III presents a refactoring example from a Class Diagram of an e-business application. The next three sections contain the basic contributions of the paper, discussing a semantics for Class Diagrams and formalising refactoring as (different kinds of) refinement with respect to the envisaged semantics. Finally, section VII concludes and points out how our results have a broader scope than just class refactoring, underlying some issues for future work.

II. Coalgebras, Bisimulation and Refinement

Given a functor T, understood as a specification of a signature of *observers*, a T-coalgebra is simply a function $p : \mathsf{T}U \longleftarrow U$ mapping elements of a state space U into their observations through T. A useful metaphor identifies functor T with a 'lens' ($\bigcirc\!\!\frown\!\!\bigcirc$), providing the unique, limited way through which the state of a system is observed. Similarly, a coalgebra $p : \bigcirc\!\!\frown\!\!\bigcirc U \longleftarrow U$ is regarded as a formal description of the observation process.

Alternatively, a T-coalgebra p can be thought of as a generalised *transition system* ${}_p\longleftarrow$, the shape of transitions being determined by T according to

$$ {}_p\longleftarrow \;\; = \;\; \in_{\mathsf{T}} \cdot p \tag{1}$$

or, introducing variables,

$$ u' \; {}_p\longleftarrow \; u \;\; \equiv \;\; u' \in_{\mathsf{T}} p \; u $$

where relation $\in_{\mathsf{T}}$ denotes structural membership [1]. In this context, the notion of *bisimulation* found in automata theory or process algebra, generalises to such T-shaped transition systems: a bisimulation is a relation over the state spaces of two coalgebras, p and q, which is *closed* for their dynamics, i.e.

$$ (x, y) \in R \;\Rightarrow\; (p\, x, q\, y) \in \mathsf{T}R \tag{2}$$

which, getting rid of variables, becomes the following inequality in the language of the (pointfree) calculus of binary relations [1]:

$$ R \subseteq p^{\circ} \cdot (\mathsf{T}R) \cdot q \tag{3}$$

where p° stands for the relational converse of p. Applying the *shunting* rule of the calculus on p°, this simplifies to

$$ p \cdot R \subseteq (\mathsf{T}R) \cdot q \tag{4}$$

Bisimilarity is entailed by coalgebra morphisms. Actually, a morphism from coalgebra q and p, is a function between their state spaces with commutes with the coalgebra dynamics, i.e., which validates the following equation:

$$ \mathsf{T}h \cdot q \;=\; p \cdot h \tag{5}$$

Again, this can be framed in terms of T-shaped transition systems:

$$ h \cdot \; {}_q\longleftarrow \;\; = \;\; {}_p\longleftarrow \cdot h \tag{6}$$

A general result in coalgebra theory asserts that the existence of a morphism between two coalgebras is enough to prove they are *bisimilar*, which provides us with a handy, calculational proof principle to verify bisimilarity.

[1]This relation coincides with datatype membership defined in [8] by a Galois connection. For the powerset functor, $\in_{\mathsf{T}}$ amounts to standard set membership, while for polynomial functors the following inductive definition applies (see [14] for details):

$$\begin{aligned}
\in_{\mathsf{Id}} &= id \\
\in_{\mathsf{K}} &= \bot \\
\in_{T_1 \times T_2} &= (\in_{T_1} \cdot \pi_1) \cup (\in_{T_2} \cdot \pi_2) \\
\in_{T_1 + T_2} &= [\in_{T_1}, \in_{T_2}] \\
\in_{T_1 \cdot T_2} &= \in_{T_2} \cdot \in_{T_1} \\
\in_{T^K} &= \bigcup_{k \in K} \in_T \cdot \beta_k \text{ (where } \beta_k f = f\ k)
\end{aligned}$$

Equation (6) is, in fact, a conjunction of inclusions

$$h \cdot {}_q\longleftarrow \;\subseteq\; {}_p\longleftarrow \cdot h \tag{7}$$

$${}_p\longleftarrow \cdot h \;\subseteq\; h \cdot {}_q\longleftarrow \tag{8}$$

which correspond to transition *preservation* and *reflection*, as the following pointwise rendering may turn more explicit:

$$v' \; {}_q\longleftarrow \; v \;\Rightarrow\; h\, v' \; {}_p\longleftarrow \; h\, v$$

$$u' \; {}_p\longleftarrow \; h\, v \;\Rightarrow\; \exists_{v' \in V} .\; v' \; {}_q\longleftarrow \; v \;\wedge\; u' = h\, v'$$

Coalgebra morphisms preserve and reflect T-shaped transitions, a basic observation which lead the authors, in a series of previous publications [13], [14], [2], to characterise a notion of coalgebraic *refinement* in terms of morphisms that only preserve or reflect such transitions. Cutting short a long story, a *forward* (respectively, *backward*) morphism is defined, with respect to a refinement preorder $\leq$, as a function $h : U_p \longleftarrow U_q$ between the state spaces of the relevant coalgebras, such that

$$\mathsf{T}\, h \cdot q \;\dot{\leq}\; p \cdot h \tag{9}$$

respectively,

$$p \cdot h \;\dot{\leq}\; \mathsf{T}\, h \cdot q \tag{10}$$

Notation $f \,\dot{\leq}\, g$ equivales to $f \subseteq \leq \cdot g$, i.e.,

$$f \,\dot{\leq}\, g \;\equiv\; \langle \forall\, x \;::\; f\, x \leq g\, x \rangle$$

As discussed extensively in the above mentioned references, $\leq$ is any preorder compatible with the membership relation, in the sense that

$$\in_{\mathsf{T}} \cdot \leq \;\subseteq\; \in_{\mathsf{T}} \tag{11}$$

i.e., for all x_1, x_2,

$$x \in_{\mathsf{T}} x_1 \;\wedge\; x_1 \leq x_2 \;\Rightarrow\; x \in_{\mathsf{T}} x_2$$

Typical refinement preorders capture reduction of non determinism and/or increase of definition, but much more possibilities can be considered (see [2] for a detailed catalogue and discussion). Reference [13] proved that forward (respectively, backward) morphisms preserve (respectively, reflect) T-shaped transitions as well as that coalgebras and such morphisms do possess, in both cases, the structure of a category.

A coalgebra q is a *forward refinement* of p, written as $q \preccurlyeq p$, if there exists a forward morphism from q to p. Dually, a *backward refinement*, written as $q \preccurlyeq p$, is witnessed by a backward morphism also from q to p.

III. A Refactoring Example

This section introduces a concrete example of refactoring over a class diagram. The original diagram is depicted in Figure 1: a simplified model of a video renting e-business.

This model, which we believe is self-explanatory, can be refactored according to the *Inline Class* refactoring pattern [6]. Consider, for example, classes *Membership*, *Account*, and *AccountItem*, which represent the clients of a store, their accounts, and the history of record items on such accounts. Each client has an account and every account has a set of

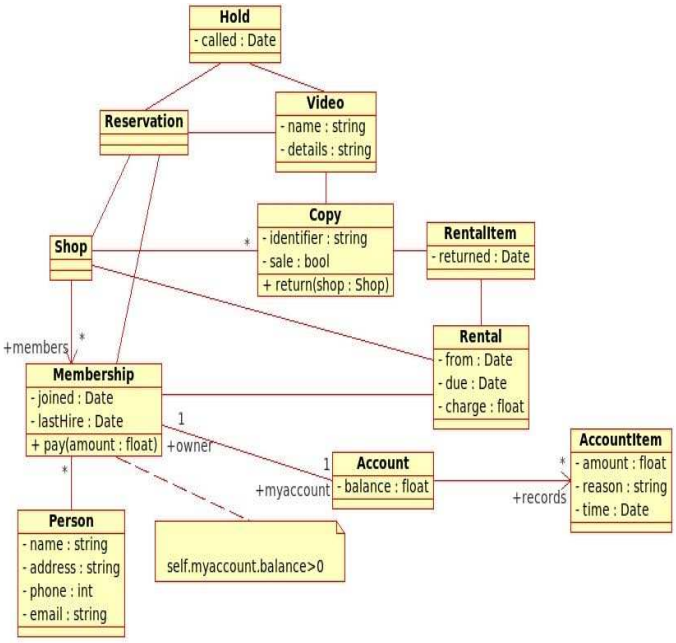

Fig. 1. Video e-business example

account recorded items. An OCL constraint is attached to class *Membership*, stating that every client's account balance is larger than 0. Class *Account* does not provide any methods of its own, being only used by class *Membership*. Therefore, classes *Membership* and *Account* can be joined into a unique class. The resulting diagram is as shown in Figure 2, where all other classes remain unchanged.

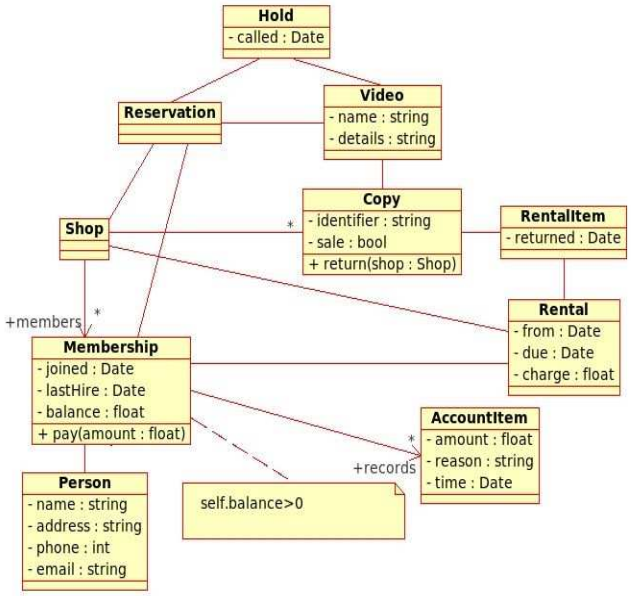

Fig. 2. Refactored class diagram

As this example shows, a refactoring is a model transformation which eventually improves its internal structure, while preserving its external behaviour, or, understanding 'external'

as an abbreviature for *externally observed*, its observational semantics. To formally record and reason about such model transformations, however, entails the need for

- a precise notion of *observational semantics* for class diagrams,
- a way to encode each refactoring pattern, or law, in the semantics and, finally,
- a proof that such encoding preserves the semantics in a stronger or weaker, but always precisely defined, way.

The rest of the paper is devoted to tackle these issues.

IV. A Semantics for Classes

A. Classes

In UML a *class diagram* captures the static structure of a system, as a set of classes and relationships, called *associations*, between them. Classes may be further annotated with *constraints*, i.e., properties that must hold for every object in the class along its lifetime. Let us concentrate, for the moment, in class declarations. The aim of a class declaration is introduce a signature of attributes and methods. Consider, for example, class **Membership** in the diagram of Fig. 1. It introduces two attributes and a method over a state space, identified by variable U below, which is made observable exactly (and uniquely) by the attributes and methods it declares. Concretely,

$$\begin{aligned}&\mathsf{joined} : Date \longleftarrow U\\ &\mathsf{lastHire} : Date \longleftarrow U\\ &\mathsf{pay} : U \longleftarrow U \times \mathbb{R}\end{aligned}$$

These three declarations can be grouped in one through a *split* construction

$$\langle \mathsf{joined}, \mathsf{lastHire}, \overline{\mathsf{pay}} \rangle : Date \times Date \times U^{\mathbb{R}} \longleftarrow U$$

which is a *coalgebra* for functor

$$\mathsf{T}\, X \;=\; Date \times Date \times X^{\mathbb{R}}$$

Therefore, we write,

$$[\![\mathbf{Membership}]\!] \;=\; \langle \mathsf{joined}, \mathsf{lastHire}, \overline{\mathsf{pay}} \rangle$$

In general, the semantics $[\![c]\!]$ of a class c is given by a specification of a coalgebra

$$\langle \mathsf{at}, \overline{\mathsf{md}} \rangle : A \times (O \times U)^I \longleftarrow U$$

where A is the attribute domain, and each method accepts a parameter, of type I, and delivers both a state change and an output value, of type O. I.e., a coalgebra for functor

$$\mathsf{T} : A \times (O \times X)^I \longleftarrow X \tag{12}$$

Typically, I and O are *sum* types, aggregating the input-output parameters of each declared method. On its turn, A is usually a *product* type joining all attribute outputs in a way which emphasises that each of them is available independent of the others, and therefore always able to be accessed in parallel.

More generally, as methods are typically implemented by *partial functions* or even by arbitrary *relations*, this definition should be generalised to

$$\langle \mathsf{at}, \overline{\mathsf{md}} \rangle : A \times \mathsf{B}(O \times U)^I \longleftarrow U$$

where B is a strong monad [2] capturing some sort of behavioural effect. For example, *partiality* (making $\mathsf{B}X = X+1$) or *non determinism* (B standing for the finite powerset functor). Additionally, a class may specify some initial conditions, typically as a predicate $\gamma : \mathbf{2} \longleftarrow U$ which is supposed to hold in the coalgebra initial states.

Such a coalgebraic setting provides for free a notion of observational equivalence — T-bisimulation —which is a fundamental tool for analysing valid refactorings. Instantiating definition (2) to functor T, yields two class models being bisimilar iff they provide identical observations through attributes and execution of the method's component not only deliver equal outputs but also makes each of them to evolve to a pair of new states which are also bisimilar.

B. Aggregating Classes in Class Diagrams

Every class in a UML Class Diagram corresponds to a coalgebraic specification (T, Φ, Ψ) in which T is the functor discussed above, which represents a generic signature of attributes and methods, Φ is a set of axioms to characterize the properties of the class, and Ψ describes the properties that hold for newly created objects.

The semantics of a class specification c in a UML Class Diagram is defined as the category of coalgebras for the corresponding coalgebraic specification and initial state preserving morphisms between them and the behaviour of the objects of class c is captured by the final coalgebra in this category. On the other hand, inheritance relationship between two classes in a class diagram is witnessed by a functor between the corresponding categories of coalgebras for the superclass and the subclass.

Such 'theory-oriented' view provides a right level of abstraction, but, on the other hand, requires some heavy machinery to be handled in its full genericity. Therefore, in the sequel, we shall concentrate into a more concrete, 'model-oriented' description, assuming a prototypical inhabitant of each class specification and defining our combinators at the model level.

[2] A *strong monad* [10] is a monad $\langle \mathsf{B}, \eta, \mu \rangle$ where B is a strong functor and both η and μ strong natural transformations. B being strong means there exist natural transformations $\mathsf{T}(\mathsf{Id} \times -) : \mathsf{T} \times - \Longleftarrow \mathsf{T} \times -$ and $\mathsf{T}(- \times \mathsf{Id}) : - \times \mathsf{T} \Longleftarrow - \times \mathsf{T}$ called the right and left strength, respectively, subject to certain conditions. Their effect is to *distribute* the free variable values in the context "$-$" along functor B. Strength τ_r, followed by τ_l maps $\mathsf{B}I \times \mathsf{B}J$ to $\mathsf{BB}(I \times J)$, which can, then, be flattened to $\mathsf{B}(I \times J)$ via μ. In most cases, however, the *order* of application is relevant for the outcome. The Kleisli composition of the right with the left strength, gives rise to a natural transformation whose component on objects I and J is given by $\delta r_{I,J} = \tau r_{I,J} \bullet \tau l_{\mathsf{B}I,J}$ Dually, $\delta l_{I,J} = \tau l_{I,J} \bullet \tau r_{I,\mathsf{B}J}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of B-computations. Whenever δ_r and δ_l coincide, the monad is said to be *commutative* and the unique transformation represented by δ.

A UML Class Diagram introduces a number of class specifications which types the object population of any corresponding model implementation. Typically, different ways of putting classes together in a Class Diagram correspond to different operators between T-coalgebras. In particular, one may consider a form of *parallel* aggregation, denoted by $\boxtimes$, in which methods in both classes can be called simultaneously (as they always act upon disjoint state spaces), and a form of *interleaving*, denoted by $\boxplus$, which offers a choice of which class to call. Note that in both cases, attributes are always available to be observed, and therefore are composed in a multiplicative context. Initial conditions are joined by logical conjunction. Therefore, given coalgebras p and q, over state spaces U and V, respectively, we define their product $p \boxtimes q$ as $\langle \gamma_{p\boxtimes q}, \langle \mathsf{at}_{p\boxtimes q}, \overline{\mathsf{md}}_{p\boxtimes q} \rangle \rangle$, where,

$$
\begin{aligned}
\gamma_{p\boxtimes q} &= U \times V \xrightarrow{\gamma_p \times \gamma_q} \mathbf{2} \times \mathbf{2} \xrightarrow{\wedge} \mathbf{2} \\
\mathsf{at}_{p\boxtimes q} &= U \times V \xrightarrow{\mathsf{at}_p \times \mathsf{at}_q} A \times A' \\
\mathsf{md}_{p\boxtimes q} &= U \times V \times (I \times I') \xrightarrow{\mathsf{m}} (U \times I) \times (V \times I') \\
&\xrightarrow{\mathsf{md}_p \times \mathsf{md}_q} \mathsf{B}(O \times U) \times \mathsf{B}(O' \times V) \\
&\xrightarrow{\delta} \mathsf{B}((O \times U) \times (O' \times V)) \\
&\xrightarrow{\mathsf{Bm}} \mathsf{B}((O \times O') \times (U \times V))
\end{aligned}
$$

where m is an isomorphism (combining Cartesian product commutativity and associativity), and δ is the Kleisli composition of left and right strengths associated to monad B. Interleaving, or *choice*, differs from $\boxtimes$ only in the methods component. Thus,

$$
\begin{aligned}
\mathsf{md}_{p\boxplus q} &= U \times V \times (I + I') \xrightarrow{\triangle \times \mathsf{id}} (U \times V)^2 \times (I + I') \\
&\xrightarrow{\cong} (U \times I) \times V + (V \times I') \times U \\
&\xrightarrow{f} \mathsf{B}(O \times U) \times V + \mathsf{B}(O' \times V) \times U \\
&\xrightarrow{\tau_r \times \tau_r} \mathsf{B}((O \times U) \times V) + \mathsf{B}((O' \times V) \times U) \\
&\xrightarrow{\cong} \mathsf{B}(O \times (U \times V)) + \mathsf{B}(O' \times (U \times V)) \\
&\xrightarrow{g} \mathsf{B}((O + O') \times U \times V) + \mathsf{B}((O + O') \times U \times V) \\
&\xrightarrow{\triangledown} \mathsf{B}((O + O') \times U \times V)
\end{aligned}
$$

where $f \stackrel{\text{abv}}{=} \mathsf{md}_p \times \mathsf{id} + \mathsf{md}_q \times \mathsf{id}$ and $g \stackrel{\text{abv}}{=} \mathsf{B}(\iota_1 \times \mathsf{id}) + \mathsf{B}(\iota_2 \times \mathsf{id})$. On the ohter hand, $\triangle = \langle \mathsf{id}, \mathsf{id} \rangle$ and $\triangledown = [\mathsf{id}, \mathsf{id}]$ denote, respectively, the diagonal and co-diagonal functions (see [4] for details on these definitions and the calculus of functions).

Finally, another tensor, denoted by $\circledast$, corresponds to what may be called *concurrent* composition. It is defined as a combination of the $\boxtimes$ and $\boxplus$, allowing for both parallel or interleaved method execution. Formally, the action of $p \circledast q$ on methods of classes p and q accepts either separated or tupled inputs to deliver the result of applying either $\mathsf{md}_{p\boxplus q}$ or $\mathsf{md}_{p\boxtimes q}$

$$
\mathsf{md}_{p\circledast q} : \mathsf{B}((O + O' + O \times O') \times U \times V) \longleftarrow U \times V \times (I + I' + I \times I')
$$

The reader can easily check $\mathsf{md}_{p\circledast q}$ is defined by

$$
\mathsf{md}_{p\circledast q} \;=\; \mathsf{B}(\mathsf{dl}^{\circ}) \cdot \delta \cdot (\mathsf{md}_{p\boxplus q} \times \mathsf{md}_{p\boxtimes q}) \cdot \mathsf{dr}
$$

where dl and dr stand for product left and right distribution, respectively.

All three combinators are associative as well as commutative, whenever B is a commutative monad. As one would expect, such properties are stated up to bisimilarity. The proof of commutativity of $\boxtimes$ below illustrates a way to reason, in a calculational style, with coalgebraic definitions.

The basic proof technique resorts to the well-known fact that a morphism between coalgebras entails bisimilarity. In this example isomorphism $\mathsf{s} : V \times U \longleftarrow U \times V$ relating the state spaces of classes $p \boxtimes q$ and $q \boxtimes p$, is shown to be a T-coalgebra morphism. The only non trivial part of the proof is the one related to the methods' component, which we detail as follows [3], in a completely pointfree style:

$$
\begin{aligned}
& \mathsf{B}(\mathsf{s} \times \mathsf{s}) \cdot \mathsf{md}_{p\boxtimes q} \\
= & \quad \{ \text{ definition of } \boxtimes \} \\
& \mathsf{B}(\mathsf{s} \times \mathsf{s}) \cdot \mathsf{Bm} \cdot \delta \cdot (\mathsf{md}_p \times \mathsf{md}_q) \cdot \mathsf{m} \\
= & \quad \{ \mathsf{B} \text{ is a functor and routine: } \mathsf{m} \cdot \mathsf{s} = (\mathsf{s} \times \mathsf{s}) \cdot \mathsf{m} \} \\
& \mathsf{B}(\mathsf{m} \cdot \mathsf{s}) \cdot \delta \cdot (\mathsf{md}_p \times \mathsf{md}_q) \cdot \mathsf{m} \\
= & \quad \{ \delta \text{ and } \mathsf{s} \text{ naturality entails } \mathsf{Bs} \cdot \delta = \delta \cdot \mathsf{s} \} \\
& \mathsf{Bm} \cdot \delta \cdot \mathsf{s} \cdot (\mathsf{md}_p \times \mathsf{md}_q) \cdot \mathsf{m} \\
= & \quad \{ \mathsf{s} \text{ naturality} \} \\
& \mathsf{Bm} \cdot \delta \cdot (\mathsf{md}_p \times \mathsf{md}_q) \cdot \mathsf{m} \cdot (\mathsf{s} \times \mathsf{s}) \\
= & \quad \{ \text{ definition of } \boxtimes \} \\
& \mathsf{md}_{p\boxtimes q} \cdot (\mathsf{s} \times \mathsf{s})
\end{aligned}
$$

Reference [5] introduces a comprehensive calculus of generalised Moore machines framed as coalgebras for a functor similar to (12) which can, to a great extent, be adapted to the present setting to reason about class specifications.

V. Refactoring Class Diagrams

The coalgebraic semantics for classes and class aggregation introduced in the previous section has the side effect of representing any parcel of a UML Class Diagram (from a single class to the whole diagram) as T-coalgebra. Moreover, the combining classes in a diagram through different tensors, entails different semantic perspectives which may be helpful in subsequent design stages. In this section, we propose to use such a framework to discuss refactoring of Class Diagrams.

[3] Note a swap of the arguments is also necessary

A. Refactoring by Refinement

Refactoring can be discussed, in practice, both at the *specification* level (in which case no particular model of any class specification in the diagram is assumed) or at the *model* level (when it is proposed with respect to a particular design model). The former, to be discussed in subsection V-B, is, certainly, more interesting and the one where typical refactoring patterns, as discussed in [6], apply. In the sequel, however, we consider a design stage where the class diagram is already under transformation towards a concrete design. In such a context, the most elementary refactoring situation captures the replacement of a particular class model by one of its refinements.

To deal with it, in our approach, is necessary to show that the class combinators which give semantics to the whole diagram preserve the refinement relation. As discussed above, a typical refinement relation captures increase in definition and reduction of non determinism — typical choices for B in T, being the *maybe* or the *powerset* monad. The following result, however, applies to *any* forward refinement $\preccurlyeq$. Moreover, a dual result can be proved, exactly along the same lines for any backward refinement $\preccurlyeq$.

Suppose, thus, that $c \preccurlyeq c'$, which means there is a forward morphism $h : c' \longleftarrow c$ such that $\mathsf{T}h \cdot c \mathrel{\dot{\leq}} c' \cdot h$. We want to prove that

$$c \boxtimes d \;\preccurlyeq\; c' \boxtimes d \tag{13}$$

The refinement situation is witnessed by a morphism between the relevant compositions which amounts to function $h \times \mathsf{id} : U' \times V \longleftarrow U \times V$. There are two inequations to prove:

$$\mathsf{at}_{c \boxtimes d} \mathrel{\dot{\leq}} \mathsf{at}_{c' \boxtimes d} \cdot (h \times \mathsf{id}) \tag{14}$$

$$\mathsf{B}((\mathsf{id} \times \mathsf{id}) \times (h \times \mathsf{id})) \cdot \mathsf{md}_{c \boxtimes d} \mathrel{\dot{\leq}} \mathsf{md}_{c' \boxtimes d} \cdot (h \times \mathsf{id}) \times (\mathsf{id} \times \mathsf{id}) \tag{15}$$

Inequality (14) is immediate from the hypothesis. For (15) we reason:

$$\begin{array}{ll} & \mathsf{B}((\mathsf{id} \times \mathsf{id}) \times (h \times \mathsf{id})) \cdot \mathsf{md}_{c \boxtimes d} \\ = & \quad \{ \text{ definition of } \boxtimes \} \\ & \mathsf{B}((\mathsf{id} \times \mathsf{id}) \times (h \times \mathsf{id})) \cdot \mathsf{Bm} \cdot \delta \cdot (\mathsf{md}_c \times \mathsf{md}_d) \cdot \mathsf{m} \\ = & \quad \{ \ \delta \text{ and } \mathsf{m} \text{ naturality} \} \\ & \mathsf{Bm} \cdot \delta \cdot (\mathsf{B}(h \times \mathsf{id}) \times \mathsf{B}(\mathsf{id} \times \mathsf{id})) \cdot (\mathsf{md}_c \times \mathsf{md}_d) \cdot \mathsf{m} \\ = & \quad \{ \ \times \text{ is a functor} \} \\ & \mathsf{Bm} \cdot \delta \cdot ((\mathsf{B}(h \times \mathsf{id}) \cdot \mathsf{md}_c) \times (\mathsf{B}(\mathsf{id} \times \mathsf{id}) \cdot \mathsf{md}_d) \cdot \mathsf{m} \\ \dot{\leq} & \quad \{ \text{ hypothesis and monotonicity; identities} \} \\ & \mathsf{Bm} \cdot \delta \cdot ((\mathsf{md}_{c'} \cdot (h \times \mathsf{id})) \times (\mathsf{md}_d \cdot (\mathsf{id} \times \mathsf{id}))) \cdot \mathsf{m} \\ = & \quad \{ \ \times \text{ is a functor; } \mathsf{m} \text{ naturality} \} \\ & \mathsf{Bm} \cdot \delta \cdot (\mathsf{md}_{c'} \times \mathsf{md}_d) \cdot \mathsf{m} \cdot (h \times \mathsf{id}) \times (\mathsf{id} \times \mathsf{id}) \\ = & \quad \{ \text{ definition of } \boxtimes \} \\ & \mathsf{md}_{c' \boxtimes d} \cdot (h \times \mathsf{id}) \times (\mathsf{id} \times \mathsf{id}) \end{array}$$

A similar result can be shown, along the same lines, for $\boxplus$ and $\circledast$. Finally, note that, as any morphism is a forward morphism as well, the result is also valid when $\preccurlyeq$ is replaced by $\sim$, i.e., for any bisimilar refactoring.

B. Refactoring Patterns

Let us re-visit the refactoring example introduced in section III to illustrate the *Inline Class refactoring* pattern which is stated as

> *Law 1:* Inline class refactoring allows two classes to be merged together provided one of them has no methods available.

As with any other refactoring pattern one would like to consider, we proceed in two steps:

- first the pattern is encoded in the semantics;
- then, it has to be shown that the original and the new diagram are observationally equivalent or else one is a refinement of the other.

Back to the example at hands, our encoding is as follows: classes **Membership** and **Account** are replaced by a new class **Membership′** whose semantics is a new coalgebra over the state space of $[\![\mathbf{Membership}]\!]$ to which a new attribute balance is added.

$$\begin{array}{l} [\![\mathbf{Membership'}]\!] \\ \quad = \langle \langle \mathsf{at}_{\mathbf{Membership}}, \mathsf{at}_{\mathbf{Account}} \rangle, \overline{\mathsf{md}}_{\mathbf{Membership}} \rangle \end{array} \tag{16}$$

We are now left with the need to record how does new class **Membership′** relates to (the relevant fragment of) the original Class Diagram, i.e.,

$$[\![\mathbf{Membership}]\!] \boxtimes [\![\mathbf{Account}]\!]$$

assuming the remaining part of the diagram remains unchanged.

Clearly, they are not bisimilar, because, at each step, the original attributes of classes **Membership** and **Account** are now computed over the *same* state.

For the methods' component, note that to assert the absence of methods declarations in class **Account** is equivalent to endow it with a unique method, with trivial argument (i.e., such that $I \cong \mathbf{1}$, where $\mathbf{1}$ is the singleton set $\{*\}$), yielding a trivial result (i.e., $O \cong \mathbf{1}$) and acting as an identity over the state space. Apart these 'dummy' parameters of type $\mathbf{1}$, no observable difference can be detected between the new class **Membership′** and the semantics of the original subdiagram containing classes **Membership** and **Account**. The latter is given by coalgebra [4]

$$\begin{array}{l} [\![\mathbf{Membership}]\!] \boxtimes [\![\mathbf{Account}]\!] \\ \quad : U \times V \;\longrightarrow\; A \times \mathsf{B}(O \times (U \times V))^I \end{array} \tag{17}$$

Clearly, projection $\pi_1 : U \longleftarrow U \times V$ is a *backward* morphism for any refinement preorder capturing increase in definition

[4] Actually, $[\![\mathbf{Membership}]\!] \boxtimes [\![\mathbf{Account}]\!]$ is from $U \times V$ to $A \times \mathsf{B}((O \times \mathbf{1}) \times (U \times V))^{I \times \mathbf{1}}$, which can be transformed into (17) by the obvious natural isomorphism.

(i.e., reduction of partiality).

$$[\![\mathbf{Membership}']\!] \cdot \pi_1 \\ \dot{\leq}\ A \times \mathsf{B}(\mathsf{id} \times \pi_1)^I \cdot [\![\mathbf{Membership}]\!] \boxtimes [\![\mathbf{Account}]\!]$$

Therefore, the *inline* refactoring pattern is actually a *backward refinement* of the new by the old diagram, i.e.,

$$[\![\mathbf{Membership}]\!] \boxtimes [\![\mathbf{Account}]\!] \preccurlyeq [\![\mathbf{Membership}']\!]$$

VI. Constraints and Associations

So far we have ignored completely the two other ingredients of a UML Class Diagram, namely, *constraints* and *associations*. The former are typically attached to class specifications and their semantic effect is to constraint what coalgebras count as valid implementation for the class. Such is the case, for example, of constraint

$$\mathsf{balance} > 0$$

attached to class **Membership** in our example.

Associations can also be interpreted as constraints, this time with respect to a fragment of the diagram containing the two associated classes. For this, one has to assume that the state space of each class has a component recording the collection of live instances. An *association* becomes a constraint over such components of the (joint) state space. For example a 'one-to-one' association corresponds to a predicate asserting the existence of an injective function relating the collection of instances of each class. Similarly, a 'one-to-many' association corresponds to a relation whose kernel is the identity, i.e., a total relation whose converse is simple.

In general, constraints and associations are predicates which are supposed to be preserved along the system life-time. Formally, they are incorporated in the semantics as *invariants*. Following the approach recently proposed in [3], such predicates, once encoded as coreflexives, i.e., fragments of the identity, according to

$$y\ \Phi_P\ x \quad \equiv \quad y = x \wedge P\, x$$

can be specified as

$$c \cdot \Phi_P \ \subseteq \mathsf{T}\, \Phi_P \cdot c \tag{18}$$

When reasoning about diagram transformations, such as refactoring, constraints entail for *proof obligations*. For example,

$$[\![\mathsf{balance} > 0]\!] \ = \\ [\![\mathbf{Membership}]\!] \cdot \Phi_{\mathsf{balance}>0} \subseteq \mathsf{T}\, \Phi_{\mathsf{balance}>0} \cdot [\![\mathbf{Membership}]\!]$$

needs to be discarded whenever justifying a refactoring involving class **Membership**.

VII. Conclusions

As announced in the introduction, this paper is just a first step on a broader attempt to apply the principles and techniques of coalgebraic semantics to reason formally about refactoring of UML models. In [17] and [15], we developed coalgebraic models for both *statecharts* and *sequence diagrams*, respectively. The former were modelled as coalgebras for

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

whereas the latter are also coalgebras for

$$\mathsf{T}X = X^\Sigma$$

over a universe of global configurations.

Preliminary work on refactoring of such diagrams indicate that, in both cases, *weaker* (i.e., *coarser*, with respect to bisimilarity) relations on the observable behaviour, namely refinement, are in order.

Just for a brief illustration, consider statechart in Figure 3 for a copy object in the video business system, which captures the following dynamics. When a copy is created, it is in some store. A copy in the store can be hired and then returned back. When hired, if the due date expires, the copy enters in an out-of-date state, and must be returned back at some time.

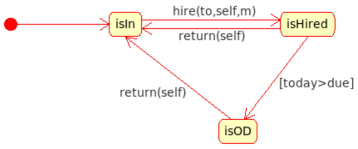

Fig. 3. Statechart for Copy

When a copy is in the store, it can be held for an outstanding reservation or put on the shelf. It is held if it is wanted, i.e., if there is a reservation for the video and in this store that does not have a Hold. If a copy is held and the reservation is cancelled, it is either reallocated to another reservation or put in the *HoldCancelled* state until checked back to the shelf. Furthermore, states *isHired* and *isOD* can be grouped together to model the behaviour of a copy when it is out of the store. With such two composite states, a refactored statechart, which actually refines the original one with respect to the semantics in [17], is represented in Figure 4.

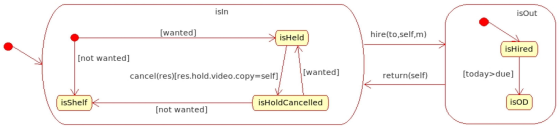

Fig. 4. Refactored Statechart for Copy

In retrospect, the approach illustrated in this paper seems promising to either to capture known refactoring patterns, or to identify new ones, for a variety of UML models. A lot of work, however, remains to be done. In particular, we would like to tackle the consistency problem among different UML view models, and to explore the relationship between model transformations and other kinds of refinement, namely, the notion of architectural refinement introduced in [16].

ACKNOWLEDGMENTS

The work reported in this paper is partially supported by a grant from the GLANCE funding program of NWO, through project CooPer (600.643.000.05N12). Thanks to the anonymous referees and to the workshop participants whose comments and questions largely helped to improve the paper.

REFERENCES

[1] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, pages 7–42. Springer Lect. Notes Comp. Sci. (755), 1993.

[2] L. S. Barbosa and J. N. F. de Oliveira. Transposing Partial Coalgebras: An exercise on coalgebraic refinement. *Theoretical Computer Science*, 365(1-2):2–22, 2006.

[3] L. S. Barbosa, J. N. Oliveira, and A. M. Silva. Calculating invariants as coreflexive bisimulations. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*, pages 83–99. Springer Lect. Notes Comp. Sci. (5140), 2008.

[4] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[5] A. Cruz, L. Barbosa, and J. Oliveira. From algebras to objects: Generation and composition. *Journal of Universal Computer Science*, 11(10):1580–1612, 2005.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] P. Harmon and M. Watson. *Understanding UML: The Developer's Guide with a Web-Based Application in Java^{TM}*. Morgan Kaufmann Publishers, Inc., 1998.

[8] P. F. Hoogendijk. *A generic theory of datatypes*. PhD thesis, Department of Computing Science, Eindhoven University of Technology, 1996.

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

[10] A. Kock. Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120, 1972.

[11] S. Marković and T. Baar. Refactoring OCL Annotated UML Class Diagrams. In *Proceedings of MoDELS 2005*, number 3713 in LNCS, pages 280–294. Springer-Verlag, 2005.

[12] S. Meng, B. K. Aichernig, L. S. Barbosa, and Z. Naixiao. A Coalgebraic Semantic Framework for Component Based Development in UML. In *Proceedings of CTCS'04*, volume 122 of *ENTCS*, pages 229–245. Elsevier Science Publishers, 2005.

[13] S. Meng and L. S. Barbosa. On Refinement of Generic State-based Software Components. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proceedings of AMAST'04*, volume 3116 of *LNCS*, pages 506–520. Springer, 2004.

[14] S. Meng and L. S. Barbosa. Components as Coalgebras: the Refinement Dimension. *Theoretical Computer Science*, 351(2):276–294, 2006.

[15] S. Meng and L. S. Barbosa. A Coalgebraic Semantic Framework for Reasoning about UML Sequence Diagrams. In *Proceedings of QSIC'08*. IEEE Computer Society, 2008.

[16] S. Meng, L. S. Barbosa, and Z. Naixiao. On Refinement of Software Architectures. In *Proceedings of ICTAC'05*, volume 3722 of *LNCS*, pages 482–497. Springer, 2005.

[17] S. Meng, Z. Naixiao, and L. S. Barbosa. On Semantics and Refinement of UML Statecharts: A Coalgebraic View. In J. R. Cuellar and Z. Liu, editors, *SEFM2004, 2nd International Conference on Software Engineering and Formal Methods*, pages 164–173. IEEE Computer Society, 2004.

[18] T. Mens. On the use of graph transformations for model refactoring. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Proceedings of GTTSE 2005*, number 4143 in LNCS, pages 219–257. Springer-Verlag, 2006.

[19] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Proceedings of 1st International Conference on Graph Transformation*, number 2505 in LNCS, pages 286–301. Springer-Verlag, 2002.

[20] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[21] Object Management Group. *OMG Model Driven Architecture*. `http://www.omg.org/mda/`.

[22] Object Management Group. *Unified Modeling Language: Superstructure - version 2.1.1*, 2007. `http://www.uml.org/`.

[23] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Framewoks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

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

[25] T. Stahl and M. Voelter. *Model Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[26] R. V. D. Straeten, V. Jonckers, and T. Mens. Supporting Model Refactoring Through Behaviour Inheritance Consistencies. In Thomas Baar et al., editor, *UML 2004*, volume 3273 of *LNCS*, pages 305–319. Springer, 2004.

[27] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel. Refactoring UML Models. In M. Gogolla and C. Kobryn, editors, *Proceedings of UML 2001*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.

[28] F. Tip, A. Kieźun, and D. Bäumer. Refactoring for generalization using type constraints. In *Object-Oriented Programming Systems, Languages, and Applications (OPPSLA 2003)*, pages 13–26, 2003.

[29] M. van Kempen, M. Chaudron, D. Kourie, and A. Boake. Towards Proving Preservation of Behaviour of Refactoring of UML Models. In *Proceedings of SAICSIT 2005*, pages 252–259, 2005.