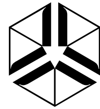


The United Nations
University

UNU/IIST

International Institute for
Software Technology

Towards a Coalgebraic Semantics of UML: Class Diagrams and Use Cases

Sun Meng and Bernhard K. Aichernig

January, 2003

UNU/IIST Report No. 272

R

UNU/IIST and UNU/IIST Reports

UNU/IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU/IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU/IIST, and
7. Dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

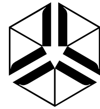
Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST at P.O. Box 3058, Macau or visit UNU/IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Chris George, Acting Director



UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Towards a Coalgebraic Semantics of UML: Class Diagrams and Use Cases

Sun Meng and Bernhard K. Aichernig

Abstract

In this paper we present a coalgebraic semantics for UML class diagrams and given some discussions on the coalgebraic semantics for use cases. A loose semantics for class diagrams is defined. The semantics of a class in UML class diagrams is given as the category of coalgebras of the corresponding class specification. Associations among classes are interpreted as coalgebraic observers. The generalization hierarchy of classes is specified by the inheritance morphism among them. Some examples on checking the internal consistency for class diagrams by exploiting the coalgebraic semantics are introduced.

Sun Meng is a Fellow at UNU/IIST on leave from the School of Mathematical Science of Beijing University, China, where he is a Ph.D candidate. His research interest include category theory, coalgebra theory, Object-Oriented method, formal method in software development, and formal semantics for modeling languages. His email address is sm@iist.unu.edu.

Bernhard K. Aichernig is a Research Fellow at UNU/IIST. He is also an assistant professor at the Institute for software Technology at the Graz University of Technology in Austria. His research interests include the synergies of testing and formal development methods, techniques of refinement, and requirements engineering supported by formal specification languages. His email address is bka@iist.unu.edu.

Copyright © 2003 by UNU/IIST, Sun Meng and Bernhard K. Aichernig

Contents

List of Figures iii

1 Introduction 1

2 A Cofibred Category of Coalgebras 3

3 Class Diagrams 4
3.1 Syntax of Class Diagram 5

4 Coalgebraic Semantics of Class Diagrams 9
4.1 Semantics of Classes 9
4.2 Semantics of Associations 18
4.2.1 Navigation 20
4.2.2 Ordering 21
4.2.3 Visibility 21
4.2.4 Changeability 22
4.2.5 Association Classes 22
4.2.6 Qualification 23
4.2.7 Aggregation and Composition 24
4.2.8 N-ary Association 25
4.3 Semantics of Generalization 27
4.3.1 Abstract Classes 31
4.3.2 Multiple Inheritance 33
4.4 Templates 33
4.5 Semantics of Class Diagrams 34
4.6 Examples in Checking Consistency of Class Diagrams 35

5 Use Cases 37
5.1 Discussions on Advanced Techniques 39

6 Related Work 43

7 Conclusion and Discussion 44

Acknowledgments 45

List of Figures

1 Different representations of a class in UML . . . 9
2 Constraint . . . 11
3 Interface . . . 13
4 Association . . . 18
5 Navigation . . . 20
6 Visibility . . . 21
7 Representation of an association class . . . 23
8 Qualification . . . 24
9 Aggregation and Composition . . . 25
10 An n-ary Association . . . 26
11 The ternary association **Record** being decomposed . . . 26
12 The decomposition of the ternary association *Record* as an association class . . . 27
13 Association being inherited . . . 30
14 Multiple inheritance with common ancestor . . . 32
15 Template classes and instantiated classes . . . 33
16 Object diagram . . . 35
17 Inconsistency of a class diagram . . . 35
18 An inconsistent class diagram for a library system . . . 36
19 A consistent class diagram . . . 37
20 Use Case Diagram . . . 37
21 Borrow Book use case . . . 38

22 Return Book use case 38
23 Pay Fee use case 39
24 RSL specification for BOOK 40
25 RSL specification for READER 41
26 RSL specification for the use cases 42

1 Introduction

Object-orientation has now become a popular approach in software industries [21, 43]. The Unified Modeling Language (UML) [45, 53, 5], which is a graphical language for specifying, visualizing, constructing and documenting object-oriented (OO) systems, has become a *de facto* standard for OO modeling. One of the main advantages of UML is that it offers a set of different view models to describe specific aspects of the system to be developed, such as the static structure of the system, the dynamic behavior of single objects in the system, and the communication and coordination between different objects in the system, etc. And these models together describe the system being designed. Moreover, the modeling languages in UML can be used for different stages of development, including requirement capture and analysis, design and implementation [39, 40]. But this also causes the main drawback of the current UML standard: the lack of a unifying formal semantics. Consequently, the definition of such a precise semantics has become an active research area.

Although the syntax of UML has been precisely defined, its semantics is still described with informal natural language in the specification. The lack of a formal semantics is now the main drawback of the current UML standard. It has been recognized that informal semantics are usually *incomplete* or *inconsistent* (or even both), and may make the interpretation of the meaning of language elements ambiguous. As a consequence, the confusion of the meaning of UML models may cause them to be used incorrectly. Therefore, the high effort being spent on modeling does not always yield the intended systems.

The separation of different view models in UML prompts one of the key questions in UML, that is, the *consistency* between diagrams representing the same information of a system. Since certain aspects of a system may be specified by more than one view model, the consistency of the family of models has to be checked to assure the correctness of the models. For example, sequence diagrams model the interaction of objects in a system while statechart diagrams model the intra behavior of single objects. Therefore, we need to check whether the objects specified by the statechart diagrams are able to satisfy the behavior requirements stated in sequence diagrams. Since different models in UML are using (paradigmatically) different languages, it will not be much helpful even we provide different formal semantics for different single models which have different semantic domains, because the consistency between models is still not obvious and the difference of semantic domains makes one can not request a common semantic interpretation as the criterion of consistency of UML models. Unfortunately, most of the previous and ongoing formalization work adopts such an approach and only focuses on individual aspects of UML and thus gives up the advantages of having multiple views. This restriction is mainly caused by the limited expressive power of the semantic domains in use. Therefore, a unifying semantics of UML is needed which uses a common semantic domain and gives interpretation for different models so that the consistency of a collection of models can be checked in the semantic domain.

Obviously, such a semantic domain must be *expressive* enough so that it can interpret different languages for both static and dynamic aspects of systems. Moreover, it should support the *composition* and *refinement* operations. The composition operation is needed to support the decomposition of a complex system into components and the connection of different components to compose them together into the whole system. For example, the semantic units representing single objects need to be composed in the semantic domain and the result should be consistent with the semantics of the systems consists of these objects.

Because UML can be used in different design stages, one model may refine the information provided by other models. For example, statechart diagrams add the dynamic behavior property of objects into class diagrams which only state their static structure. Such refinement also need to be supported in the semantic domain to check the consistency between a concrete model and a more abstract model which specify the same aspects of a system.

In order to use UML more effectively in software development, during our work on formalizing UML, we found that coalgebraic structures [31, 54] provide a powerful semantic domain which is able to cover different aspects of software system. Coalgebra theory, which is a relatively new research field, has been recognized as a suitable framework for the description of state-based dynamic systems such as automata, transition systems and classes in object-oriented languages, where observations and behavior patterns are more relevant than data construction. In Peter Aczel's foundational work on the theory of "non well-founded sets" [2], a coalgebraic approach is used to describe non-deterministic transition systems and construct a model for non well-founded sets. Later works by Bart Jacobs and others on coalgebraic specification and coalgebraic semantics for object oriented programming (see e.g. [23, 24, 25, 28, 50, 48, 61]) has proved that coalgebras are suitable for modeling dynamic systems, especially, for classes in object oriented programming languages. H. Tews makes an extension to the polynomial functors used by Jacobs so that binary methods can also be represented by coalgebras [61]. [32] presents the assertional and behavior refinement of coalgebraic specifications. A coalgebraic class specification language CCSL is also developed recently [52]. In order to fully and natually capture both computational and observational aspects of systems, The works on integrating algebraic and coalgebraic techniques in specification of systems by taking a layered approach can be found in [9, 10].

By resorting the underlying *environment category* to a set based category enriched with some algebraic structure, we can naturally get a more concrete semantics for UML specifications. In fact, just by varying the environment category, coalgebras can describe different computation models as discussed in [**?**]. The obvious benefits of such a coalgebraic semantics are more straightforward notions of consistency and refinement between different kinds of UML diagrams. In this paper we propose a method for specifying and reasoning about UML models of software systems based on the coalgebraic semantics and first-order logic. This is the first step towards such a unifying semantics of the UML. Since coalgebras support the dynamic aspects quite naturally, we concentrate here on the static models represented by class diagrams. Some form of familiarity with category theory and coalgebra theory are assumed for the reader, and we will not explain all the notions in this paper in elementary terms.

In this paper, we define the coalgebraic semantics of UML class diagrams based on our earlier work on the coalgebraic calculus for systems in [60] and give some discussion on the formalization of use cases via coalgebraic specifications as the first step towards a unifying semantics for UML. The semantic domain is defined via a cofibred category of coalgebras. We give the semantics of UML elements based on the theory of coalgebra and define the consistency relation between them in the coalgebraic interpretation. The central idea of our coalgebraic semantics is that a set of UML diagrams denote coalgebraic specifications as introduced by Jacobs [24, 25, 29]. More precisely, the presented coalgebraic semantics translates the graphical symbols and annotations of the various UML diagrams into functors and properties of a coalgebraic specification. With this approach, standard definitions in coalgebraic contexts, like bisimilarity and refinement [32] can be also applied to UML diagrams. Another obvious advantage of our coalgebraic approach is that it covers both the static models and the dynamic models of UML, which

makes the definition of consistency between different models natural in the semantic domain. Moreover, it also provides the static models (e.g. class diagrams) a behavior semantics which is helpful for understanding of the evolution of system in the early phases of system development. The aim of our work is to provide a coalgebraic semantic framework suitable for the notations in UML and which can form a basis for rigorous object-oriented software development.

This paper is organized as follows: Section 2 is a brief introduction to the cofibred category of coalgebras as the semantic domain. An informal syntax for class diagrams is presented in Section 3. In Section 4, a coalgebraic semantics for class diagrams is defined, and some examples on checking the consistency of class diagrams are given. We discuss the use cases in Section 5 by using the coalgebraic specifications to describe use cases. Some related work are given in Section 6. Finally, Section 7 concludes and show the future work.

2 A Cofibred Category of Coalgebras

In this section we briefly present our semantic domain. It is designed to unify the different semantic aspects of UML. It turns out that a cofibred category of coalgebras is the most promissing candidate for our endeavor. The relatively heavy mathematical machinery pays off in providing the necessary concepts of abstraction, and thus in simplifying our semantic definitions. Advantages of such an approach include: (1) The theory of universal coalgebras have been proven useful in modeling dynamic and static aspects of object-oriented systems faithfully since we have the freedom to choose the signature functor appropriately [54]. (2) The notion of functors in category theory provides a powerful theory of interfaces and signatures. (3) Our cofibrated category $\mathbf{C_F}$ represents a category which contains the transition structures corresponding to different functors and allows us to relate coalgebras of different functors within just one category.

Let $\mathbf{Set}$ be the category of sets and functions. The functors considered in this paper map $\mathbf{Set}$ to $\mathbf{Set}$ (endofunctors on the category $\mathbf{Set}$), and together with the natural transformations between them form a category $[\mathbf{Set}, \mathbf{Set}]$. These functors are used to describe the coalgebraic signatures $\alpha : U \to FU$ that map a state $u : U$ to its possible observations $\alpha(u) : FU$. For every such an endofunctor F, we can obtain the category $\mathbf{C}_F$ of all F-coalgebras.

The cofibred category $\mathbf{C_F}$ based on a subcategory $\mathbf{F}$ of $[\mathbf{Set}, \mathbf{Set}]$ is defined as the "total" category which encompasses the categories $\mathbf{C}_F$ for all the possible functors F in $\mathbf{F}$, which is a subcategory of $[\mathbf{Set}, \mathbf{Set}]$, and also admits natural transformations as morphisms between different functors. Consequently, the semantic domain provides the mapping between different coalgebras, and thus between our interpretations of different UML diagrams. The following construction defines the cofibred category of coalgebras:

Proposition 2.1 *Let* $\mathbf{F}$ *be a subcategory of the category* $[\mathbf{Set}, \mathbf{Set}]$ *of endofunctors on the category* $\mathbf{Set}$*. Then for two endofunctors* F *and* G *in* $\mathbf{F}$*, a natural transformation* $\eta : F \to G$ *allows us to view every* F*-coalgebra* $\alpha : U \to F(U)$ *as a* G*-coalgebra* $\eta_U \circ \alpha : U \to G(U)$*. Take coalgebras for functors in* $\mathbf{F}$

as objects and for two coalgebras $(U, \alpha : U \to F(U))$ and $(V, \beta : V \to G(V))$, take (σ, η) as an arrow between them, where σ is function between their carrier and $\eta : F \Rightarrow G$ a natural transformation, such that the following diagram commutes:

$$\begin{array}{ccccc} U & & \xrightarrow{\quad\sigma\quad} & & V \\ \alpha\downarrow & & & & \downarrow\beta \\ F(U) & \xrightarrow{\eta_U} & G(U) & \xrightarrow{G(\sigma)} & G(V) \end{array}$$

This defines a category $\mathbf{C_F}$ which is a cofibration over the category $\mathbf{F}$.

Proof: The composition of two arrows $f_1 = (\sigma_1, \eta_1) : c \to d$ and $f_2 = (\sigma_2, \eta_2) : d \to e$ is $f_2 \circ f_1 = (\sigma_2 \circ \sigma_1, \eta_1 \circ \eta_2) : c \to e$. Associativity of composition is inherited from $\mathbf{Set}$ and $\mathbf{Cat}$. It is even easier to show that there is an identity morphism for any coalgebra $c = (U, \alpha : U \to F(U))$ which is defined by (id_U, id_F) where id_U is the identity function on the state space U and id_F is the identity natural transformation from F to itself. Clearly $\mathbf{C_F}$ is a cofibration over the category $\mathbf{F}$ where we have a functor $p : \mathbf{C_F} \to \mathbf{F}$ which maps every F-coalgebra to the functor F and every arrow (σ, η) to η.

Here we recall some standard terminology in category theory [4, 26]. For a total category $\mathbf{C_F}$ and the base category $\mathbf{F}$ of this cofibration, an object c of $\mathbf{C_F}$ and an arrow $f : c \to d$ with $pc = F$ and $pf = \eta$ are called *over* F and *over* η respectively.

Note that all the functors we use in the sequel are in a particular collection of functors: the so-called *Kripke polynomial functors* (KPFs). Such functors are the endo-functors on $\mathbf{Set}$ finitely built up from the following syntax:

$$F(X) ::= C \mid X \mid F_1(X) \times F_2(X) \mid F_1(X) + F_2(X) \mid C \to F_1(X)$$

where C is an arbitrary non-empty constant set, F_1, F_2 are two previously defined KPFs. The *product* of two sets A, B is a set $A \times B$ in $\mathbf{Set}$ together with two projections $\pi_1 : A \times B \to A, \pi_2 : A \times B \to B$. The *coproduct (sum)* of A and B is an object $A + B$ together with two injections $\iota_1 : A \to A + B, \iota_2 : B \to A + B$. The *exponent* A^B (or $B \to A$) is used for the collection of functions from a set B to A.

3 Class Diagrams

One of the main artifacts to produce in OO modelling are class diagrams. A class diagram shows the static structure of a system, consisting of a set of classes and the relationships between them. A class is an abstract description of a set of objects with similar structure, behavior and relationships. The description of a class includes the common attributes and operations of the objects belonging to the class, whereas the structural relationships between it and other classes are represented by generalizations and associations, including aggregation and composition.

3.1 Syntax of Class Diagram

The UML Specification [45] describes the UML abstract syntax as UML class diagrams depicting the UML metamodel and well-formedness rules together with examples. There are descriptive informal text associated with each diagram for describing the abstract syntax.

Here we give the brief description of the abstract syntax of UML class diagrams in a top-down fashion, the detailed formal syntax and well-formedness rules for class diagrams defined by using RSL can be found in [15]. A class diagram is formed by classes and relationships among them. A set of well-formed rules is needed to assure that the class diagram is well-formed. Before we give the definition of class diagram, we first introduce a set of names $\mathbf{Name}$ as the name space of a class diagram, the elements in which are the names of classes, attributes, operations and associations. We use $\mathcal{T}$ for the family of data types, every element in which corresponds to a set of data values. For example, $\mathbf{Bool}$ and $\mathbf{Int}$ are used for boolean and integer types.

Definition 3.1 (Class Diagram) *A class diagram is a tuple:* $CD = \langle \mathbb{C}, \mathbb{A}, \lhd\!\!\longleftarrow, WF_{CD} \rangle$ *where*

- $\mathbb{C}$ *is a nonempty finite set of classes;*
- $\mathbb{A}$ *is a finite set of associations;*
- $\lhd\!\!\longleftarrow \subseteq \mathbb{C} \times \mathbb{C}$ *is the generalization relationship between classes;*
- WF_{CD} *is a set of well-formedness rules on* CD*.*

A class is a description of a set of objects that share the same attributes, operations and relationships. In UML class diagrams, a class consists of a name, a set of attributes, a set of operations and a multiplicity. A list of properties may be listed in a class to show some class attributes or tagged values. A class can be abstract, root or leaf in the generalization relationship. A template is a class with one or more formal parameters which describes a family of classes, each class specified by binding the parameters to actual values.

Definition 3.2 *A class* $\mathbf{C}$ *consists of the following parts:*

- *the name of the class* $name(\mathbf{C}) \in \mathbf{Name}$*;*
- *the set of its attributes* $Ats(\mathbf{C})$*;*
- *the set of its operations* $Ops(\mathbf{C})$*;*
- *its multiplicity* $m(\mathbf{C}) \subseteq \mathbf{Int}$ *which is optional;*
- *the optional* $isAbstract : \mathbf{Bool}$ *which specifies whether it can be directly instantiated;*

- *the optional* $isRoot$: $\mathbf{Bool}$ *which specifies whether it has no ancestors;*
- *the optional* $isLeaf$: $\mathbf{Bool}$ *which specifies whether it has no descendants;*
- *the optional parameter list* P *provided to a parameterized class (template).*

An attribute describes a range of values that instances of a class may hold. It has a visibility, a name, a type, a multiplicity which is the possible number of data values for the attribute that may be held by one object, a changeability to show whether the attribute value may be changed after the object is created, an initial value specifying the attribute value upon initialization, and a target scope specifying whether the targets are Instances or Classifiers.

Definition 3.3 *An attribute* $\mathbf{At}$ *of class* $\mathbf{C}$ *consists of the following parts:*

- *the optional visibility of the attribute* $visibility(\mathbf{At})$ *which may be* $public$*,* $private$ *or* $protected$*;*
- *the name of the attribute* $name(\mathbf{At})$*;*
- *the type of the values of the attribute* $type(\mathbf{At})$*, which may be basic types or other classes in the class diagram;*
- *the optional multiplicity of the attribute* $m(\mathbf{At}) \subseteq \mathbf{Int}$*;*
- *the optional changeability of the attribute* $changeability(\mathbf{At})$ *which may be* $frozen$*,* $addOnly$ *or* $changeable$*;*
- *the optional initial value of the attribute* $in(\mathbf{At})$ *which has type* $type(\mathbf{At})$*;*
- *the optional target scope of the attribute* $scope(\mathbf{At})$ *which may be* $classifier$ *or* $instance$*.*

The default syntax given in UML specification [45] is:

visibility name [multiplicity]: type-expression = initial-value $\{$*property-string*$\}$

An operation is a service that can be requested from an object to effect behavior. It has a visibility, a name, a signature which consists of a list of formal parameters and an optional result type. Each element of the parameter list has a name and a type. Furthermore, an operation may have a scope and can be abstract.

Definition 3.4 *An operation* $\mathbf{Op}$ *of class* $\mathbf{C}$ *consists of the following parts:*

- *the optional visibility of the operation* $visibility(\mathbf{Op})$ *which may be* $public$*,* $private$ *or* $protected$*;*
- *the name of the operation* $name(\mathbf{Op})$*;*

- *the list of its formal parameters* $P_{\mathbf{Op}}$*, the elements of which has the form*

$$kind\ parameter_name : parameter_type = default_value,$$

  where $kind$ *include* **in**, **out** *and* **inout**, *the default value is* **in**. *The type of* $P_{\mathbf{Op}}$ $type(P_{\mathbf{Op}})$ *is the product of the types* $parameter_type$ *of its elements;*

- *the optional result type* $rtype_{\mathbf{Op}}$*;*

- *the optional scope of the operation* $scope(\mathbf{Op})$ *to show if its applicable to the instances of the class or the class itself;*

- *the optional* $isAbstract : \mathbf{Bool}$ *which specifies whether the implementation of the operation is supplied in the class or by a descendant.*

- *the optional* $isQuery : \mathbf{Bool}$ *states whether or not it will change the state of the object after applying the method.*

The default syntax of operation is:

visibility name (parameter-list): return-type-expression {*property-string*}

Different classes in a class diagram are related by different kinds of relationships. Relationships in UML class diagrams are separated into three categories: associations, generalizations and dependencies. Each relationship should satisfy the well-formedness rules.

Definition 3.5 *A relationship is an association, a generalization or a dependency among classes together with a set of well-formedness rules on it.*

An association in a class diagram describes discrete connections among objects or other instances in a system [53]. An association may have a name and two or more association ends, each of them specifies a class being connected by the association and a set of optional properties that must be fulfilled for the relationship to be valid [45], including a role name, a multiplicity, a navigability, an aggregation property, a changeability, an ordering property, a target scope and a visibility.

Definition 3.6 *An association* $\mathbf{A}$ *consists of the following two parts:*

- *the name of the association* $name(\mathbf{A})$*;*
- *the set of association ends* $\{e : End(\mathbf{A})\}$.

Definition 3.7 *An association end* $e : End(\mathbf{A})$ *of association* $\mathbf{A}$ *consists of the following parts:*

- *the class being connected at this end* $class(e)$*;*
- *the role name of the end* $name(e)$*, which provides a name for traversing from a source instance across the association to the target instance or set of target instances when it is put on a target end;*
- *the multiplicity of the end* $m(e) \subseteq \mathbf{Int}$ *which specifies the number of target instances that may be associated with a single source instance through the association when it is put on a target end;*
- *the navigability* $isNavigable(e) : \mathbf{Bool}$ *which specifies whether traversal from a source instance to its associated target instances is possible when it is put on a target end;*
- *the aggregation property* $AggregationKind(e)$ *specifies whether the end is an aggregation with respect to another end, the possible values of it include* $aggregate$*,* $composite$ *and* $none$*;*
- *the changeability* $changeability(e)$ *which specifies whether an instance of the association may be modified from another end, the possible values of it include* $frozen$*,* $addOnly$ *or* $changeable$*;*
- *the ordering* $ordering(e)$ *which specifies whether the set of links from the source instance to the target instance is ordered when it is put on a target end, the possible values of it include* $unordered$ *and* $ordered$*;*
- *a target scope* $scope(e)$ *which specifies whether the target value is an instance or a classifier;*
- $visibility(e)$ *specifying the visibility of the end from the viewpoint of the class on the other end, possible values are* $public$*,* $private$ *and* $protected$*.*

A generalization is a relationship between a superclass and a subclass, in which objects of the subclass are substitutable for objects of the superclass.

Definition 3.8 *A generalization* $\mathbf{C}_1 \lhd\!\!\longleftarrow \mathbf{C}_2$ *is a directed relationship between two classes: a subclass* $\mathbf{C}_2$ *and its superclass* $\mathbf{C}_1$*.*

A dependency is a relationship from a client to a supplier which states that the client requires the presence and knowledge of the supplier. According to [5], all relationships, including association, generalization are kinds of dependencies. Different stereotypes can be defined to represent shades of dependencies, and each stereotype has its own semantics. Therefore, it is almost impossible to give a precise semantics to such a relationship. Its semantics is decided by the kind of dependency being used by the users. In this paper, we only focus on the other kinds of relationships and discuss their semantics in the following.

A well-formed class diagram should satisfy a set of well-formedness rules being given in [45]. A RSL representation of the formal syntax and well-formedness rules of UML class diagrams can be found in [15]. In the following, we always assume that the class diagrams are well-formed if not explicitly specified.

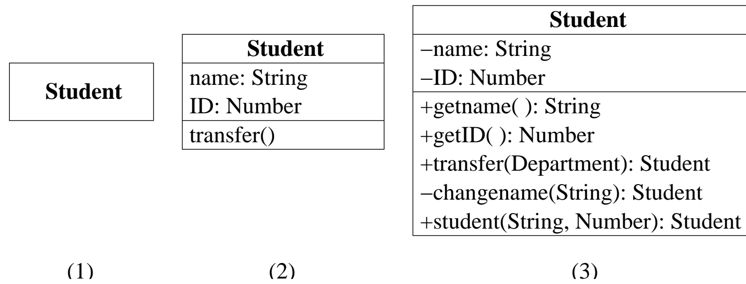


Figure 1: Different representations of a class in UML

4 Coalgebraic Semantics of Class Diagrams

4.1 Semantics of Classes

In UML, a class is a description of a set of objects that share the same attributes, methods and relationships. Every object o of a class $\mathbf{C}$ in a system has an identifier id_o which is unique in the system. We denote the set of identifiers as Id. Therefore, an object $o : \mathbf{C}$ is represented as a triple $o = (id_o, U_{\mathbf{C}}, \alpha_{\mathbf{C}} : U_{\mathbf{C}} \to F(U_{\mathbf{C}}))$ where $U_{\mathbf{C}}$ is the state space of the class $\mathbf{C}$, and F 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, state space $U_{\mathbf{C}}$ and transition structure $\alpha_{\mathbf{C}}$ remain the same. Therefore, we can use a pair (id_o, u) to present an object o at a particular state u.

In different development phases, either or both of the attribute and operation compartments may be shown when needed and omitted in other contexts. See Figure 1 as an example for a class with details suppressed, details in analysis-level and in design-level.

For the most abstract form of class representation in which all the details are suppressed, a class is interpreted as a set of objects, which assigns a type to the class. For example, the first class in Figure 1 defines a class **Student** as a set $Student$. An object of this class $s : \mathbf{Student}$ is interpreted as an element of this set $s \in Student$. Such a set is called the *object type* of the class [1].

Using the form of semantic function, we can define the semantics of a class with only one name and details suppressed.

$$\mathcal{S}[\![\mathbf{C}]\!] \triangleq C$$

where C is a set of all the objects of the class. Every element in C has an identifier and a hidden state space of possible states and will be instantiated by a coalgebra in later stages of development when the attributes and methods of the class are specified.

At the analysis level of software development, the type of attributes and methods of a class may be shown in the class diagram. See Figure 1 (2) as an example. From the object-oriented perspective, classes are built around a hidden state space, so we have the following definition:

Definition 4.1 (coalgebraic class signature) *A coalgebraic class signature Σ is a pair $(X, \alpha : X \to F(X))$ where X is a carrier set corresponding to the state space of the class, and F is a polynomial functor which is a product of a finite set of functors $F = \prod_i F_i$. $\alpha : X \to F(X)$ is used to represent the types of the attributes and methods in the class. It is also sufficient to give the pair (X, F) to represent the class signature as an abbreviation.*

Consider Figure 1 (2), we can get its signature as a pair $(Student, F)$ where $Student$ is the carrier set, and $F(X) = String \times Number \times X^{Department}$. [1] In this example X corresponds to the carrier set $Student$. Thus the sequence of its methods (attributes) $\langle name, ID, transfer \rangle$ can be identified with a function $Student \to F(Student)$.

Therefore, a UML class **C** in this stage gives a class signature (C, F) and can be interpreted as a category of coalgebras of such a functor F.

$$\mathcal{S}[\![\mathbf{C} \mid at1 : A_1, \ldots, at_m : A_m, op_1 : B_1 \to C \times D_1, \ldots op_n : B_n \to C \times D_n]\!] \triangleq \mathbf{Coalg}_{F_\mathbf{C}}$$

where F_C is an abbreviation for the functor $F_C : X \to \prod_{1 \le j \le m} A_j \times \prod_{1 \le i \le n} (X \times D_i)^{B_i}$. $\mathbf{Coalg}_{F_C}$ is the category of F_C-coalgebras (as objects) and F_C-homomorphisms (as arrows) [2].

Note that here we do not distinguish between attributes and methods, like e.g. in Eiffel. [24] shows that both attributes and methods can be represented by functors in a unified form. The difference between them lies in the form of the functors being used. For the attributes, the associated functor is a constant functor which does not change the state space. For methods, the associated functor does affect the state space of the class. Moreover, we do neither give the visibility description of attributes and methods, nor their implementation details. In fact, these should be given at a later phase of development, as the third class in Figure 1 shows, or specified in other diagrams (such as statechart diagrams). The attributes such as $name$ should be private and can not be used or modified outside the class (as shown in the third class in Figure 1), that means, they are not same as the ordinary observers because they can only be used inside the hidden state space. Therefore, we can get the intuition of separating the observers of a class into external and internal parts to represent the public view and private view of the class separately. Moreover, there will be another view from the perspective of generalization: protected view, which includes the methods that can be accessed by the class and its subclasses. This separation is very useful for the designer of a system, and will be discussed below.

Now we turn to such classes at the design level. UML provides three kinds of visibility for the attributes and methods of a class at this phase: public(+), protected(#) and private(-).

The public view of a class describes its public methods by which the state space of the class can be visited and modified from outside. Therefore, we can get the following definition of an external class signature.

[1] the exponential X^Y represents the set of arrows from Y to X.

[2] Note that here we make a simplification and only use polynomial functors for signatures of the methods. For binary methods or more generally, n-ary methods which takes both covariant and controvariant appearance of the object type of the class, a framework is provided by Hendrik Tews which uses extended polynomial functors. Readers can use [61, 62] as references. Another point is that the category is a subcategory of $\mathbf{C_F}$ in Section 2

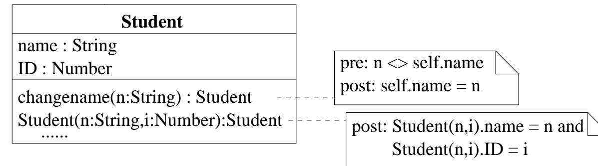


Figure 2: Constraint

Definition 4.2 (external class signature) *An external class signature is a pair* (X, F_{pub}) *in which:*

- X *describes the carrier set of the class;*
- F_{pub} *is a functor on the state space* X*, being used to represent the public view of the class.*

As an example, the external class signature of the third class in Figure 1 is the pair $(Student, F_{pub})$ where $F_{pub}(X) = String \times Number \times X^{Department}$ The public method $student$ is used for creating new objects of this class. It can not be used as a component of F_{pub} because it is not an observer but a creator (usually being called constructor in OO method). Methods like this can be modeled by the concept of *initial state* in the state space of the coalgebra which is used by Jacobs in the definition of the semantics of objects and classes [23, 24].

A user of a class is not interested in the details of the class's internal implementation, but only in the behavior it can provide. In fact, this is why the coalgebraic description is appropriate for classes. In coalgebraic approaches, the state space of a class is dealt with as a black box which can only be accessed via specified operations which represent the functions of the class. However, from the developer's perspective, what is inside the state space of a class also should be considered as well as the public part of the class in order to implement (additional) methods of the class. Similarly as the definition of external class signature, we can get the definition of signatures (X, F_{pri}) and (X, F_{pro}) where F_{pri} and F_{pro} are functors for private and protected views respectively.

Now we can get the definition of (internal) class signature which describes the internal structure and behavior of classes as follows:

Definition 4.3 (internal class signature) *An (internal) class signature is a pair* (X, F) *where* X *is the carrier set being used to specify the class state space and* F *is a functor which is the product of functors* $F = F_{pub} \times F_{pro} \times F_{pri}$ *being used to represent the types of all the attributes and methods provided in the class.*

In fact, a class signature is not enough for specifying a class in UML (especially in the later phases). In a class diagram, the values of some attributes and the behavior of some methods may be specified explicitly by constraints. The concept of constraint allows semantics of some modeling elements to be specified linguistically. UML provides a constraint language OCL [45] to describe such constraints. The

constraints in OCL are invariants and pre- and post-conditions which can be used to state conditions that must be satisfied for any models.

The constraints on a class in UML can be represented as a set of formulas Φ.Since Moss [44] first realized that the shape of a coalgebra (its interface functor) determines a logical modal language, a lot of work on the connection of coalgebras and temporal logic followed [27, 51, 35, 36]. We can also use the associated temporal operators in the formulas in Φ naturally and thus every axiom can be a formula. However, first-order logic is enough here. For example, the constraint for method $changename$ in Figure 2 can be translated to the following axiom:

$$\forall s : Student, n : String.(\neg(n = name(s)) \Rightarrow name(changename(s, n)) = n)$$

The constraint for method $Student$ in Figure 2 is used for specifying the initial attribute value of newly created objects of this class. Such conditions for newly created objects also can be represented as a set Ψ of formulas. As an example, the constraint in Figure 2 is described as:

$$\forall n : String, i : Number.(name(student(n, i)) = n \wedge ID(student(n, i)) = i)$$

In the semantic definition of such classes in UML class diagrams, we follow the classic work of Jacobs and others [23, 24, 25, 50] on the coalgebraic semantics for classes. Every class in a UML class diagram is taken as a coalgebraic specification $\mathbf{Spec}$.

Definition 4.4 (class specification) *A class specification is a tuple* (F, Φ, Ψ) *in which:*

- F *is a functor on a local state space* X*, being used to represent all the attributes and methods of the class;*
- Φ *is a set of axioms that gives the constraints to the functors for the attributes and methods to characterize the properties of the class;*
- Ψ *describes the properties that hold for newly created objects.*

A model (class implementation) of a given class specification $\mathbf{Spec} = (F, \Phi, \Psi)$ is a triple $c = (U, \alpha : U \to F(U), u_0)$, where U is a carrier set interpreting the state space of the class, $\alpha : U \to F(U)$ 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 (not abstract) class $\mathbf{C}$ in a UML class diagram is defined as the category $\mathbf{Coalg}(\mathbf{Spec})$ of models of the corresponding coalgebraic class specification $\mathbf{Spec}$ together with the initial state preserving homomorphisms between them.

$$\mathcal{S}[\![\mathbf{C}]\!] \triangleq \mathbf{Coalg}(F_{\mathbf{C}}, \Phi_{\mathbf{C}}, \Psi_{\mathbf{C}}) \qquad \text{if } isAbstract(\mathbf{C}) = False$$

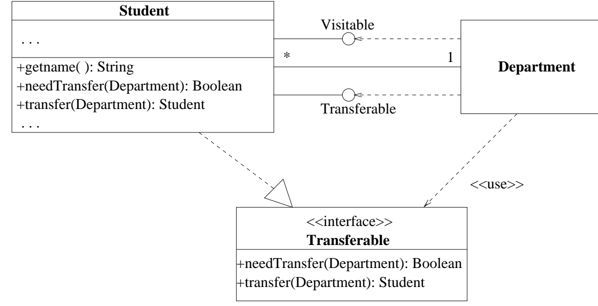


Figure 3: Interface

where $(F_{\mathbf{C}}, \Phi_{\mathbf{C}}, \Psi_{\mathbf{C}})$ is the specification of class $\mathbf{C}$ and the boolean value function $isAbstract$ specifies whether the class $\mathbf{C}$ can be directly instantiated. This category is a subcategory of $\mathbf{C_F}$ defined previously. Its objects are

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

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. The arrows in the category are initial state preserving $F_{\mathbf{C}}$-homomorphisms.

An interface in UML class diagrams is the description of the externally visible behavior of a class without specifying the internal structure. Usually, each interface contains only a part of the operations of a class and no attributes. Figure 3 is an example of an interface. The interface **Transferable** specifies a part of the behavior of class **Student** which can be viewed by class **Department**. The following definition shows the specification of a class interface.

Definition 4.5 (class interface specification) *A class interface specification* $\mathbf{Spec}_{ext}$ *of a class specification* $\mathbf{Spec} = (F, \Phi, \Psi)$ *is a pair* (F_{ext}, Φ_{ext}) *in which:*

- F_{ext} *is a functor on the state space of* $\mathbf{S}pec$*, being used to represent an interface of the class;*
- Φ_{ext} *is a set of axioms that gives the constraints to the functor* F_{ext} *to characterize the externally observable properties of the class interface;*

Given an interface specification $\mathbf{Spec}_{ext} = (F_{ext}, \Phi_{ext})$ of $\mathbf{Spec}$, for a model $c = (U, \alpha : U \to F(U))$ of $\mathbf{Spec}$, a corresponding interface is $c_{ext} = (U, \alpha_{ext} : U \to F_{ext}(U))$ which is a restriction $c_{ext} = c \upharpoonright F_{ext}$ and satisfies all the properties given by Φ_{ext}.

We can find that Definition 4.5 is somewhat similar as Definition 4.4, but there is no conditions for newly created objects. This is because "an interface is formally equivalent to an abstract class . . . "[45] and we can not create an object from an abstract class.

In the previous definitions, F_{ext} is used to represent an interface of a class, while F describes the structure and behavior of the class which is useful for the designer and programmer of the class but may not be visited from outside. The following definition describes the relationship between these two specifications, which also gives the interpretation of encapsulation.

Definition 4.6 (encapsulation) *Let* $\mathbf{Spec} = (F, \Phi, \Psi)$ *and* $\mathbf{Spec}_{ext} = (F_{ext}, \Phi_{ext})$ *be a class specification and one interface specification of this class respectively, then an encapsulation from* $\mathbf{Spec}$ *to* $\mathbf{Spec}_{ext}$ *is a set of restrictions* $f = (id, \beta)$ *between each implementation of* $\mathbf{Spec}$ *and its restriction under* $\mathbf{Spec}_{ext}$*, where the natural transformation* $\beta : F \Rightarrow F_{ext}$ *is a projection from* F *to* F_{ext} *defined as follows:*

$$\beta = \prod_{\prod_i F_i = F_{ext}} (\pi_i : F \to F_i)$$

such that for all $\phi \in \Phi_{ext}$ *and* $c = (U, \alpha, u_0)$ *being a model of* $\mathbf{Spec}$*,* $c \models \phi$*.*

From Definition 4.6 we can get the following proposition:

Proposition 4.1 *If there is an encapsulation from* $\mathbf{Spec} = (F, \Phi, \Psi)$ *to* $\mathbf{Spec}_{ext} = (F_{ext}, \Phi_{ext})$*, then* $c_{ext} = (U, \alpha_{ext} : U \to (\beta \circ F)(U))$ *is an interface for every model* $c = (U, \alpha : U \to F(U), u_0)$ *of* $\mathbf{Spec}$*.*

The relationship of a class c and its interface c_{ext} can be expressed by the following commuting diagram.

$$\begin{array}{ccc} U & \xrightarrow{c} & F(U) \\ & \searrow^{c_{ext}} & \downarrow \beta_U \\ & & F_{ext}(U) \end{array}$$

Intuitively, the transition structure α, α' for two implementations $c = (U, \alpha : U \to F(U), u_0)$ and $c' = (U', \alpha' : U' \to F(U'), u'_0)$ of the same class specification $\mathbf{Spec} = (F, \Phi, \Psi)$ should be equal. Therefore, we have a F-homomorphism f between them:

$$\begin{array}{ccc} U & \xrightarrow{f} & U' \\ \downarrow \alpha & & \downarrow \alpha' \\ F(U) & \xrightarrow[F(f)]{} & F(U') \end{array}$$

and $f(u_0) = u'_0$.

The identity function on a $\mathbf{Spec}$-model is always a F-homomorphism preserving initial state, and the composition of two F-homomorphisms preserving initial state is still a F-homomorphism preserving initial state. Thus the models of a specification $\mathbf{Spec} = (F, \Phi, \Psi)$ (which are F-coalgebras satisfying Φ and Ψ) together with the F-homomorphisms preserving initial state forms a category, which is denoted by $\mathbf{Coalg}(\mathbf{Spec})$. This is a subcategory of the category $\mathbf{C}_F$ for the restriction of Φ and Ψ on the possible state spaces.

Definition 4.7 (semantics of a class) *Every class in a UML class diagram is a coalgebraic specification. The semantics of a class in a UML class diagram is the category* $\mathbf{Coalg}(\mathbf{Spec})$ *of models of the corresponding coalgebraic class specification* $\mathbf{Spec}$ *together with the initial state preserving homomorphisms between them.*

Remark. From Theorem 2.5 in [54], we know that the graph of the homomorphism f between any two models of a same class specification is a bisimulation between them. Thus we can say that they are bisimilar (observational equivalent). That means, we can have different implementations for one class specification given by a UML class diagram and they are not distinguished from outside and one such implementation can be used instead of another.

Now we can get the following semantic function for the items in UML classes[3]:

For attributes, the default syntax is:

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

The semantic function of an attribute At in class $\mathbf{C}$ is defined as follows:

$$\mathcal{S}[\![v\ At:\ T\ [m] = i\{p\}]\!] \triangleq \{At : U_{\mathbf{C}} \rightarrow F_{At}(U_{\mathbf{C}}) \mid \mathcal{S}[\![v]\!] \wedge \mathcal{S}[\![At[m]]\!] \wedge \mathcal{S}[\![At = i]\!] \wedge \mathcal{S}[\![At\{p\}]\!]\} \quad (1)$$

where F_{At} is the functor $F_{At} : U_{\mathbf{C}} \rightarrow \mathcal{P}(\mathcal{S}[\![T]\!])$ where $\mathcal{P}$ is the powerset functor used for multiplicity of the attribute (can be dropped whenever the multiplicity is exactly one), and v is used for visibility of the attribute At:

- $\mathcal{S}[\![v = +]\!] \triangleq F_{At} \subseteq F_{pub}$;
- $\mathcal{S}[\![v = \#]\!] \triangleq F_{At} \subseteq F_{pro}$;
- $\mathcal{S}[\![v = -]\!] \triangleq F_{At} \subseteq F_{pri}$;

The *multiplicity* part shows the multiplicity of the attribute, which can be omitted, in which case it is exactly one (1..1). The semantic function for multiplicity of an attribute At in class $\mathbf{C}$ is:

$$\mathcal{S}[\![At[m]]\!] \triangleq \forall (U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0 \in U_{\mathbf{C}}) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]), \forall u \in U_{\mathbf{C}}.card(At(u)) = m$$

[3] We assume that for a data type T, $\mathcal{S}[\![T]\!] \triangleq [\![T]\!]$, $[\![T]\!]$ is the set which includes all the values of the type.

and if the multiplicity is specified as a range

$$\mathcal{S}[\![At[l..k]]\!] \triangleq \forall(U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0 \in U_{\mathbf{C}}) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]), \forall u \in U_{\mathbf{C}}.l \leq card(At(u)) \leq k$$

The additional *ordering* property is meaningful if the multiplicity upper bound is greater than one. The values may be unordered or ordered, the default value is unordered:

$$\mathcal{S}[\![At[l..k\ ordered]]\!] \triangleq l > 1 \wedge \forall u \in U_{\mathbf{C}}.At(u) \text{ is an ordered set}$$

The *initial value* is used for initializing the attribute of a newly created object, it can be omitted. Its semantic function is:

$$\mathcal{S}[\![At = i]\!] \triangleq \forall(U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]).At(u_0) = i;$$

The optional *property string* indicates property values of the attribute, like changeability. We give the semantics of the changeability and scope properties of attributes as follows:

- $\mathcal{S}[\![At\{frozen\}]\!] \triangleq \forall(U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]), \forall u \in U_{\mathbf{C}}.At(u) = At(u_0);$
- $\mathcal{S}[\![At[l..k]\{addOnly\}]\!] \triangleq k > l \wedge \forall(U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]), \forall u_1, u_2 \in U_{\mathbf{C}}.(u_1 \stackrel{\alpha_{\mathbf{C}}}{\rightarrow} u_2) \Rightarrow At(u_1) \subseteq At(u_2);$
- $\mathcal{S}[\![At\{changeable\}]\!] \triangleq \forall(U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]).\exists u_1, u_2 \in U_{\mathbf{C}}.u_1 \neq u_2 \wedge At(u_1) \neq At(u_2);$
- $\mathcal{S}[\![\underline{At}]\!] \triangleq \forall(U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]), \forall o_1, o_2$ be objects of class $\mathbf{C}$, (u_1, u_2) is a state in the state space of $o_1 \boxtimes o_2.\mathcal{S}[\![At]\!](u_1) = \mathcal{S}[\![At]\!](u_2);$[4]
- If the scope of an operation is instance (the default), then the semantic function is as (1) shows: $\forall(U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]), \forall u \in U_{\mathbf{C}}.At(u) \in \mathcal{S}[\![T]\!].$

For operation compartment of a class, the default syntax is:

visibility name(*parameter-list*) : *return-type-expr*{*property-string*}

The parameter-list part is a list of formal parameters, each element is specified with the syntax:

kind name : *type-expression* = *default-value*

where *kind* is **in**, **out** or **inout**, with the default **in** if absent. The semantic function of an operation Op in class $\mathbf{C}$ is defined as follows:

$$\mathcal{S}[\![v\ Op(i_1 : I_1, \ldots, i_n : I_n, \mathbf{out}\ o_1 : O_1, \ldots, \mathbf{out}\ o_m : O_m, \mathbf{inout}\ b_1 : B_1, \ldots, \mathbf{inout}\ b_k : B_k) : A\{p\}]\!]$$
$$\triangleq \{Op : U_{\mathbf{C}} \rightarrow F_{Op}(U_{\mathbf{C}}) \mid \mathcal{S}[\![v]\!] \wedge \mathcal{S}[\![Op\{p\}]\!]\} \quad (2)$$

[4] which is the UML notation used for representing that the scope of *At* is the classifier and each value contains a reference to the target Classifier itself but not to an Instance of the Classifier. A classifier scope attribute corresponds to the static attributes in object oriented programming languages such as C++, which means that the value of this attribute is same for all the objects of the class at the same time. In general, such attributes should be private, shared among a set of objects and with the guarantee that no other objects can have access to that attribute.

where F_{Op} is the functor $F_{Op}(U_{\mathbf{C}}) = \mathsf{Bh}(U_{\mathbf{C}} \times \mathcal{S}[\![O \times B \times A]\!])^{\mathcal{S}[\![I \times B]\!]}$ representing the signature of the method Op, $I = \Pi_{i=1}^{n} I_i$, $O = \Pi_{i=1}^{m} O_i$, $B = \Pi_{i=1}^{k} B_i$ are used for the types of the input, output and inout parameters respectively. A is the type of the returned value. Bh is the behavior monad presenting the behavior pattern of the method, and v is the visibility of the method, which is similar as in the semantics of attributes:

- $\mathcal{S}[\![v = +]\!] \triangleq F_{Op} \subseteq F_{pub}$;
- $\mathcal{S}[\![v = \#]\!] \triangleq F_{Op} \subseteq F_{pro}$;
- $\mathcal{S}[\![v = -]\!] \triangleq F_{Op} \subseteq F_{pri}$;

We now consider the semantics of the property-string which specifies the property of a method. In the following, we give the semantics of the query and scope properties.

- $\mathcal{S}[\![Op\{query\}]\!] \triangleq \forall (U_{\mathbf{C}}, \alpha_{\mathbf{C}}, u_0) \in Obj(\mathcal{S}[\![\mathbf{C}]\!]), \forall u \in U_{\mathbf{C}}, p : I \times O \times B, Op(u)(p) = (u, a)$ where $a \in \mathcal{S}[\![O \times B \times A]\!]$;
- If the scope of an operation is instance (the default), then the semantic function is as (2) shows;
- If the scope of an operation is the classifier[5], then the corresponding function is not on any instance of the class, but on the class, and does not need to be invoked for a particular object of the class. Suppose Op is a classifier scope operation in class $\mathbf{C}$, then it can only change the static attributes of this class.

  $$\mathcal{S}[\![\underline{v\ Op(pl) : A\{p\}}]\!] \triangleq \mathcal{S}[\![v\ Op(pl) : A\{p\}]\!]$$

  which satisfies that at any possible state $u \in U_{\mathbf{C}}$ of an object o of class $\mathbf{C}$, *pl* be a list of parameters, $u' = \pi_1(Op(u, pl))$ be the successor state of u after the execution of the classifier scope operation Op, then for all instance scope attribute At, $At(u') = At(u)$.

Theorem 4.1 *The execution of a static operation in class* $\mathbf{C}$ *keeps the bisimulation relationship between two objects of this class.*

Proof: The proof of this theorem is easy. Suppose for two objects o_1 and o_2 of class $\mathbf{C}$, $\approx$ is a bisimulation relationship between them. Let u_1, u_2 be two states corresponds to the two objects separately, $u_1 \approx u_2$, u'_i is the successor state of u_i for $i = 1, 2$ after the execution of a classifier operation Op, then we have for all instance scope attributes A_I, $A_I(u'_i) = A_I(u_i), i = 1, 2$, so $A_I(u'_1) = A_I(u'_2)$ because $A_I(u_1) = A_I(u_2)$. From the semantics of classifier scope attributes, we can know that for any classifier scope attribute $A_{\mathbf{C}}$, $A_{\mathbf{C}}(u_i)$ be same for all objects at the same time. Therefore o_1 and o_2 are equivalent for all observations after the execution of Op.

[5] Such an operation corresponds to a static method in C++.

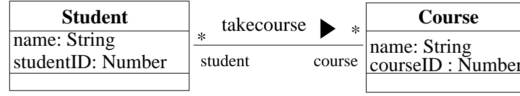


Figure 4: Association

This theorem simplifies the construction process of bisimulation between objects similar as that we discussed for components in [59]. We only need to consider the instance scope operations of a class. The result can be very useful, for example, it can reduce the complexity of verification of properties for a system in the process of model checking and improve the efficiency of test case generation in testing.

The properties *isAbstract, isLeaf* and *isRoot* which are related to inheritance of classes will be discussed in section 4.3.

Finally, we will show the semantics for interfaces. From previous discussion we can know that an interface is a collection of operations that specify a service provided by a class (in UML interfaces can also be used for specifying services of components). Therefore, an interface must be attached to the class that realizes the interface. So the semantic function for an interface **I** of class **C** is:

$$\mathcal{S}[\![<< Interface >> \mathbf{I}]\!] \triangleq \{I : U_{\mathbf{C}} \rightarrow F_{\mathbf{I}}(U_{\mathbf{C}}) \mid (U_{\mathbf{C}}, \alpha : U_{\mathbf{C}} \rightarrow F_{\mathbf{C}}(U_{\mathbf{C}}), u_0) \in Obj(\mathcal{S}[\![\mathbf{C}]\!])\}$$

where $F_{\mathbf{I}} \subseteq F_{\mathbf{C}}$ describes the operations to represent the behavior of the interface.

4.2 Semantics of Associations

An association in a class diagram describes the connections among objects in a system. It may have two or more association ends. In this paper, we first take into account the binary association. The interpretation for n-ary associations is given in section 4.2.8. Figure 4 shows an example of association.

Suppose $\mathbf{Spec_U}$ and $\mathbf{Spec_V}$ are two class specifications of class **U** and **V** in a class diagram. **A** is a binary association between them. $c = (U, \alpha)$ and $d = (V, \beta)$ are objects in $\mathbf{Coalg}(\mathbf{Spec_U})$ and $\mathbf{Coalg}(\mathbf{Spec_V})$ respectively, then association **A** which connects the two coalgebras (classes) can be interpreted as a state 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 being linked by the association simultaneously at the state s. Every pair of objects is called a link between them. The state s also provides a global view for the whole system consists of the two classes associated by **A**.

Every association has three basic components: a name, the role and the multiplicity at each of its ends. The semantic for an association is interpreted by the corresponding observers in each of the classes being related by the association. We will give the semantic function for associations as follows:

For an association **A** between class **U** and **V** in a class diagram (sometimes the association name **A** is omitted), $\mathbf{Spec_U}$ and $\mathbf{Spec_V}$ are two class specifications corresponding to the two classes. The role

names on the two ends are u_A and v_A. The multiplicities on the two ends are m_U and m_V, which are two sets of non-negative integers. Then the semantics of an association is defined as a pair of the coalgebraic observers:

$$(u_A : (Id \times V) \to \mathcal{P}(Id \times U),\ v_A : (Id \times U) \to \mathcal{P}(Id \times V)) \quad (3)$$

where U and V are the statespaces of the coalgebras $(U, \alpha : U \to F_U(U), u_0)$ and $(V, \beta : V \to F_V(V), v_0)$ which are the objects in the categories corresponding to the semantics of classes **U** and **V**.

In fact, for an association which is given as the pair of observers in (3), we can find that they are observers on objects of the two involved classes and do not change their states. Therefore, they can be treated as attributes in the corresponding classes. So the attributes of a class can be separated into two categories: the *value* attributes which corresponds to the attributes inside the class and the *reference* attributes for the associations of the class.

In order to represent the association between two classes, the two coalgebraic observers must be related as the following law:

Law 4.1 *For all objects o_U, o_V of classes* **U** *and* **V**, *we have*

$$o_U \in u_A(o_V) \ \Leftrightarrow\ o_V \in v_A(o_U).$$

So the semantic function of an association is given as the pair of observers in (3) which satisfying **Law 4.1**:

$$\mathcal{S}[\![\mathbf{U}^{u_A} \underset{\mathbf{A}}{\text{—}}^{v_A} \mathbf{V}]\!] \triangleq \{(u_A, v_A) \mid \mathbf{Law4.1}\}$$

The UML specification [45] 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”.

The multiplicity property of an association end which specifies how many objects of a class at the given end can be linked with a single object of another class can be described by the cardinality restriction of the range sets of the corresponding coalgebraic observer. For example, the multiplicity $*$ on the **Student** end in Figure 4 means that one course can be taken by any number of students and the multiplicity $*$ on the **Course** end means that one student can take any number of courses. There is no restriction on the upper bound of the multiplicity. So we can represent these conditions as:

$$\forall c : Course.card(student(c)) \geq 0$$
$$\forall s : Student.card(course(s)) \geq 0$$

where $Course$ and $Student$ represent the set of courses and students at any system state, $student$ and $course$ are the observers corresponding to the association **takecourse**.

If the multiplicity is presented as a pair $lb..ub$ where lb and ub correspond to the lower bound and upper bound separately, then such an association will be interpreted as:

$$\mathcal{S}[\![\mathbf{U}^{u_A}\underset{\mathbf{A}}{\text{---}}{}^{v_A}_{lb..ub}\mathbf{V}]\!] \triangleq \{(u_A, v_A) \mid \mathbf{Law4.1} \wedge \forall o_U : Id \times U.card(v_A(o_U)) \geq lb \wedge card(v_A(o_U)) \leq ub\}$$

where v_A is the role name on the association end at $\mathbf{V}$ side.

In general, the multiplicity at an association end can be stated as $s = lb_1..ub_1, lb_2..ub_2, \ldots, lb_n..ub_n$, which is a sequence of pairs, then the association will be interpreted as:

$$\mathcal{S}[\![\mathbf{U}^{u_A}\underset{\mathbf{A}}{\text{---}}{}^{v_A}_{s}\mathbf{V}]\!] \triangleq \{(u_A, v_A) \mid \mathbf{Law4.1} \wedge \\ \forall o_U : Id \times U. \bigvee_{i=1,\ldots,n} (card(v_A(o_U)) \geq lb_i \wedge card(v_A(o_U)) \leq ub_i)\}$$

From the discussions above for Figure 4, we can find that once the multiplicity is given explicitly in the diagram, the semantic function is:

$$\mathcal{S}[\![\mathbf{U}^{u_A}_{m_U}\underset{\mathbf{A}}{\text{---}}{}^{v_A}_{m_V}\mathbf{V}]\!] \triangleq \{(u_A, v_A) \mid \mathbf{Law4.1} \wedge (\forall o_U : Id \times U.(card(v_A(o_U)) \in m_V)) \wedge \\ (\forall o_V : Id \times V.(card(u_A(o_V)) \in m_U))\}$$

There are a number of properties can be used to model the details of a system, such as navigation, qualification, and constraints on associations. We will discuss them in detail in the following separately.

4.2.1 Navigation

For a plain association such as that in Figure 4, it is possible to navigate from objects of any class to objects of the other. In other words, navigation across the association is bidirectional. However, sometimes the navigation is limited to just one direction. See Figure 5 as an example which describes the services of an operation system. Given a $User$, the corresponding $Password$ objects can be found, but given a $Password$, the corresponding $User$ can not be identified.

Figure 5: Navigation

For an association $\mathbf{A}$ between classes $\mathbf{U}$ and $\mathbf{V}$ in a class diagram, the navigation is represented as $\mathbf{U} \rightarrow \mathbf{V}$. The other conditions are the same as in the definition of general associations. Then the semantics of navigation is given as:

$$\mathcal{S}[\![\mathbf{U}\underset{\mathbf{A}}{\longrightarrow}{}^{v_A}_{m_V}\mathbf{V}]\!] \triangleq \{v_A : Id \times U \rightarrow \mathcal{P}(Id \times V) \mid \forall o_U : Id \times U.card(v_A(o_U)) \in m_V\}$$

Figure 6: Visibility

4.2.2 Ordering

Usually, an observer for an association yields an unordered set of objects. However, sometimes ordering constraints are added to association ends to state whether the objects related to a single object at the other end of this association have an order that should be preserved. In the example of Figure 5, the $Password$s associated with a $User$ object may be kept in a least-recently used order and be marked as $ordered$.

In fact, the ordering constraint is "a performance optimization and is not an example of a logically ordered association" [46]. The multiplicity must be greater than 1 so that the objects being connected to one object at another side can form an order. The semantics of an ordered association is, that a total order $<$ must be considered on the set of observed objects (without duplicates):

$$\begin{aligned}\mathcal{S}[\![\mathbf{U}^{u_A}_{m_u} \underset{\mathbf{A}}{\text{---}} {}^{v_A\ ordered}_{m_v}\mathbf{V}]\!] \triangleq & \{(u_A, v_A) \mid \mathbf{Law 4.1} \wedge (\forall o_U : Id \times U.(card(v_A(o_U)) \in m_V))\wedge \\ & (\forall o_V : Id \times V.(card(u_A(o_V)) \in m_U)) \wedge \forall n \in m_v.n > 1 \wedge \\ & \exists < : V \times V.\ \textbf{isTotalOrder}(<)\}\end{aligned}$$

where for an object o_U of class **U**, $v_A(o_U)$ is the set of objects of class **V** that can form a sequence according to a particular order. Note that this is a formal definition of the "is an ordered set" property previously shown for attributes.

4.2.3 Visibility

The visibility of an association end can be specified as that of an attribute or method in a class by appending a visibility symbol to the role name of the end. See Figure 6 as an example.

There are three levels of visibility in UML for association end. The default kind is public, which means other classes may navigate the association and use the role name similar to the use of a public attribute. Private visibility indicates that only the class at the other end may navigate the association and use the role name, and objects at the end are not accessible to any objects outside the association. Protected visibility means that only descendants of the class at the other end may access the association and use the role name. Since the semantics of an association is given by attributes in the classes being related,

the semantic functions for visibility of association end are similar as those of the visibility of attributes:

$$\mathcal{S}[\![+u_A]\!] \triangleq u_A \in F_{pub}$$
$$\mathcal{S}[\![\#u_A]\!] \triangleq u_A \in F_{pro}$$
$$\mathcal{S}[\![-u_A]\!] \triangleq u_A \in F_{pri}$$

4.2.4 Changeability

The changeability property of association ends is similar to that of attributes. The default changeability is *changeable* in which the links can be added, deleted and moved and no indicator is needed to be given in the diagram. If the changeability is marked as *frozen*, then no links may be added, deleted or removed from an object since its creation and initialization. If the changeability is marked as *addOnly*, then links may be added to an object, but may not be deleted or modified.

The semantic function of changeability for an association end is defined as follows:

$$\mathcal{S}[\![\mathbf{U}^{u_A} \underset{\mathbf{A}}{—}^{v_A\ frozen} \mathbf{V}]\!] \triangleq \{(u_A, v_A) \mid \mathbf{Law 4.1} \wedge \forall o_V = (id_{o_V}, v) : Id \times V.\pi_1(u_A(o_V)) \equiv \pi_1(u_A(id_{o_V}, v_0))\}$$

where v is any possible state of object o_V, v_0 is its initial state and

$$\begin{aligned}\mathcal{S}[\![\mathbf{U}^{u_A}_{m_U} \underset{\mathbf{A}}{—}^{v_A\ addOnly}_{m_V} \mathbf{V}]\!] \triangleq \{(u_A, v_A) \mid\ & \mathbf{Law 4.1} \wedge (\forall o_U : Id \times U.(card(v_A(o_U)) \in m_V)) \wedge \\ & (\forall o_V : Id \times V.(card(u_A(o_V)) \in m_U)) \wedge card(m_U) > 1 \wedge \\ & \forall o_V = (id_{o_V}, v) : Id \times V.\pi_1(u_A(o_V)) \subseteq \pi_1(u_A(id_{o_V}, v')) \\ & \text{for any successor state } v' \text{ of } v\}\end{aligned}$$

Here the operation π_1 is the projection from a pair to its first component: $\pi_1(a, b) = a$. The result of the application of π_1 to a set of pairs is the set of results by applying π_1 to every element.

4.2.5 Association Classes

UML allows an association to have its own attributes, which is represented by an association class. An association class is an association that is also a class. It defines a set of features to the association itself but not to any of the classes. Generally, we can decompose an association class into a class and two one-to-many associations to represent it. See Figure 7 as an example, we can describe the association class by the class **Job** and the two associations *has* and *for*. And because *has* is one-to-many and *for* is one-to-one, we can get the result that the cardinality of the range sets of *provider*, *owner* and *jobof* can

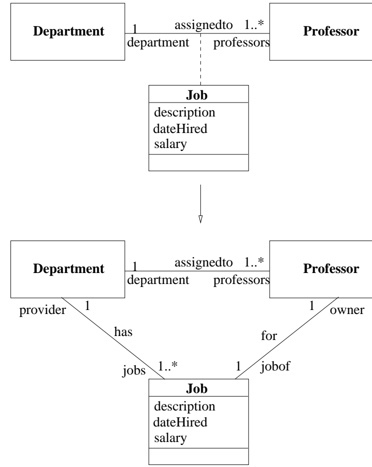


Figure 7: Representation of an association class

only be 1. Therefore, we can flatten the powersets in the ranges and get the observers equally as follows:

$$
\begin{aligned}
&\mathit{provider} : Job \rightarrow Department^{\Sigma};\\
&\mathit{jobs} : Department \rightarrow \mathcal{P}(Job)^{\Sigma};\\
&\mathit{owner} : Job \rightarrow Professor^{\Sigma};\\
&\mathit{jobof} : Professor \rightarrow Job^{\Sigma}
\end{aligned}
\qquad (4)
$$

The result of the observers of association *assignedto* at a specified system state σ can be got from (4) as follows:

$$
\begin{aligned}
&department(p, \sigma) = provider(jobof(p, \sigma), \sigma)\\
&professors(d, \sigma) = \{owner(j, \sigma) \mid j \in jobs(d, \sigma)\}
\end{aligned}
$$

The semantic function of an association class is defined by the semantics of the class together with the semantics of two one-to-many associations as follows:

$$
\begin{aligned}
\mathcal{S}[\![\mathbf{U}^{u_V}_{m_U} \underset{\mathbf{AC}}{\overline{\quad\quad}} {}^{v_U}_{m_V}\mathbf{V}]\!] \triangleq \{ & (c, asso_U, asso_V) \mid c \in \mathcal{S}[\![\mathbf{AC}]\!] \wedge \\
& asso_U \in \mathcal{S}[\![\mathbf{U}^{u_{AC}}_{1} \text{—} {}^{a_U}_{m_V}\mathbf{AC}]\!] \wedge asso_V \in \mathcal{S}[\![\mathbf{AC}^{a_V}_{m_U} \text{—} {}^{v_{AC}}_{1}\mathbf{V}]\!] \wedge \\
& \forall o_U : Id \times U, o_V : Id \times V . o_V \in v_U(o_U) \Leftrightarrow \exists! a : A . o_V = v_{AC}(a) \wedge a = a_U(o_U)\}
\end{aligned}
$$

4.2.6 Qualification

An association can be qualified. A qualifier is an association attribute or a tuple of attributes whose values partition the set of objects related to an object across an association. See Figure 8 as an example

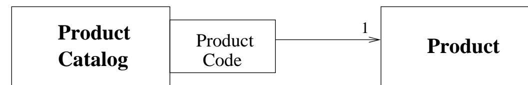


Figure 8: Qualification

of qualification where we model an association between two classes $\mathbf{ProductCatalog}$ and $\mathbf{Product}$. In the context of $\mathbf{ProductCatalog}$, we have a $\mathbf{ProductCode}$ which identifies a particular $\mathbf{Product}$. In this sense, $\mathbf{ProductCode}$ is an attribute of the association. Given an object of $\mathbf{ProductCatalog}$ and given a particular $\mathbf{ProductCode}$, we can navigate to zero or one object of $\mathbf{Product}$. Therefore, we can interpret an association with a qualifier as an association class similar as the previous section and interpret the qualifier as an attribute of the class.

Therefore the semantic function of qualification is defined as:

$$\mathcal{S}[\![\mathbf{U}(\mathbf{Q})\text{—}\mathbf{V}]\!] \triangleq \mathcal{S}[\![\mathbf{U}\underset{\mathbf{Q}}{\text{—}}\mathbf{V}]\!]$$

4.2.7 Aggregation and Composition

A binary association may represent a whole-part relationship in UML, which is called an *aggregation*. Simple aggregation is a kind of association which is "entirely conceptual and does nothing more than distinguish a 'whole' from a 'part'." [5] and *composition* is "a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. ..." [46]. Figure 9 shows an example containing one aggregation, two composition and one ordinary association which is taken from [17]. Every paper has one or more authors, exactly one abstract and one or more sections, and can be associated with at most one conference.

In fact, the definition of aggregation and composition via meta attributes instead of using two meta classes specifying their own characteristics separately in [45] and the next version of UML in [46] is argued to be unclear and dubious by many researchers [20, 8, 56]. Brian Henderson-Sellers *et al.* have done a lot of comprehensive work on analyzing the precise semantics of different kinds of WPRs (whole-part relationship) [20, 8]. Contradiction in the definition of aggregation about whether the parts can be removed from the whole before its death is noted explicitly in [8] and a revised metamodel of relationship is introduced where the WPR is defined independently from associations, and aggregation and composition forms two disjoint subclasses of WPR.

According to [8], a WPR must be asymmetry at instance level (an object can not be directly or indirectly a part of itself, $o \neq (o, -)$). There is at least one property of a whole object is independent of its parts' properties and at least one property of the whole object whose value is determined by its parts. The distinction between aggregation and composition is made clear by what are called "Secondary Characteristics" in [8, 20], such as shareability, seperability/mutability, lifetime dependency, existential dependency, and so on.

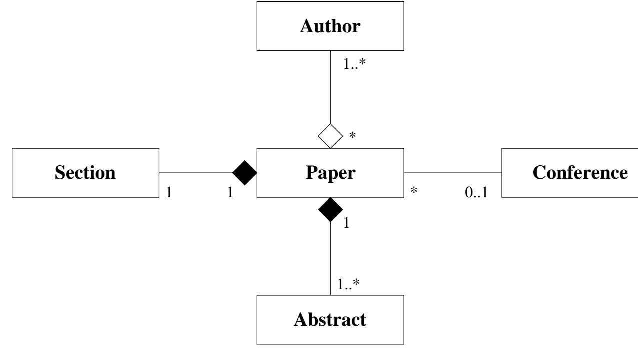


Figure 9: Aggregation and Composition

An object in an aggregation can be used as part of more than one objects. However, in a composition, an object may be a part of only one composite at a time.

$$\begin{aligned}
\mathcal{S}[\![\mathbf{U}^{u_A}_{m_U} \diamond\!\!-\!\!- \,{}^{v_A}_{m_V}\mathbf{V}]\!] \triangleq & \{a \mid a \in \mathcal{S}[\![\mathbf{U}^{u_A}_{m_U} - {}^{v_A}_{m_V}\mathbf{V}]\!] \wedge (m_U > 1 \Rightarrow \\
& \exists v \in V.\exists o_1, o_2 : Id \times U.((u_1, u_2) \text{ is a state of } o_1 \boxtimes o_2 \wedge \\
& u_1 = (v, -) \wedge u_2 = (v, -)))\} \\
\mathcal{S}[\![\mathbf{U}^{u_A}_{m_U} \blacklozenge\!\!-\!\!- \,{}^{v_A}_{m_V}\mathbf{V}]\!] \triangleq & \{a \mid a \in \mathcal{S}[\![\mathbf{U}^{u_A}_{m_U} \diamond\!\!-\!\!- \,{}^{v_A}_{m_V}\mathbf{V}]\!] \wedge 0 \leq m_U \leq 1 \wedge \\
& \forall v : V.((\exists u_1, u_2 : U.u_1 = (v, -) \wedge u_2 = (v, -)) \Rightarrow \\
& u_1 = u_2 \text{ is the state of the same object of class } U)\}
\end{aligned}$$

4.2.8 N-ary Association

An n-ary association is an association among three or more classes. In this section, we establish an approach to semantics for such associations which is a generalization of the binary case. An instance of a n-ary association is an n-tuple of objects of the respective classes.

The multiplicity of an n-ary association is less obvious than binary multiplicity. The multiplicity on one end represents "the potential number of instance tuples in the association when the other N-1 values are fixed" [45].

In order to illustrate our approach, consider the ternary association in Figure 10 without the association class $Record$, which shows the record of a team in each season with a particular goalkeeper that may be traded during the season and can be in different teams. This association is denoted by the following observations where $Year$, $Player$ and $Team$ are the corresponding set of objects: :

$$\begin{aligned}
&\textit{team} : Year \times Player \rightarrow \mathcal{P}(Team); \\
&\textit{season} : Team \times Player \rightarrow \mathcal{P}(Year); \\
&\textit{goalkeeper} : Year \times Team \rightarrow \mathcal{P}(Player);
\end{aligned}$$

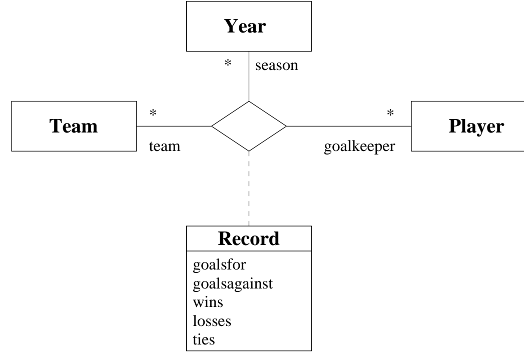


Figure 10: An n-ary Association

and all of them must satisfy the multiplicity constraint given in the diagram.

Our approach is based on a decomposition of the association into three binary associations, as shown in Figure 11. Each of the three new binary associations links one of the classes with the *Cartesian product class* of the others.

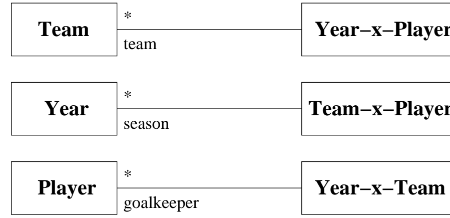


Figure 11: The ternary association **Record** being decomposed

A Cartesian product class $\prod_{1 \leq i \leq n} \mathbf{C}_i$ is an auxiliary class, whose objects are n-tuples of objects of its component classes $\mathbf{C}_i$. The semantics of such a tuple of objects is the parallel composition of the corresponding coalgebras $c_i \in \mathcal{S}[\![\mathbf{C}_i]\!]$. More precisely, parallel composition is defined by the free product operator $\otimes$ (see [60] for its formal definition). Thus, formally we have

$$\mathcal{S}[\![\prod_{1 \leq i \leq n} \mathbf{C}_i]\!] \triangleq \mathbf{Coalg}_{\times}$$

where $\mathbf{Coalg}_{\times}$ is a category with objects defined as free products of coalgebras c_i denoted by $\bigotimes_{1 \leq i \leq n} c_i$, whose action is the parallel composition of the component coalgebras.

The multiplicity constraints on the observations are:

$$\forall y : Year, p : Player.card(\mathit{team}(y,p)) \geq 0;$$
$$\forall t : Team, p : Player.card(\mathit{season}(t,p)) \geq 0;$$
$$\forall y : Year, t : Team.card(\mathit{goalkeeper}(y,t)) \geq 0;$$

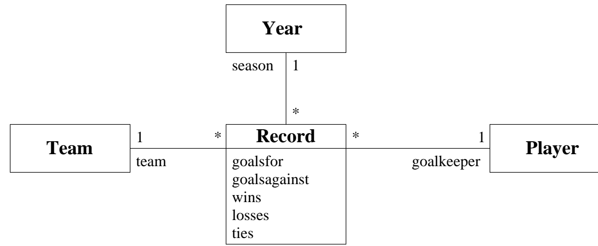


Figure 12: The decomposition of the ternary association *Record* as an association class

If an n-ary association has its own attributes, such as the class $Record$ in Figure 10, we can decompose it into three binary associations between the involved n classes and the association class as Figure 12 shows and get three pairs of observations.

4.3 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 $\mathbf{D}$ being a subclass of $\mathbf{C}$ in a class diagram is represented as $\mathbf{C} \lhd\!\!\!-\!\!\!- \mathbf{D}$. We also say that $\mathbf{D}$ inherits from $\mathbf{C}$. If $\mathbf{D}$ inherits from $\mathbf{C}$, then all the public and protected attributes and methods in $\mathbf{C}$ can be found in $\mathbf{D}$, either same as in the superclass or being overloaded. Moreover, all the axioms and the creating properties in $\mathbf{C}$ should be satisfied in $\mathbf{D}$, and may be strengthened.

If there is such an inheritance relationship between $\mathbf{D}$ and $\mathbf{C}$, a forgetful functor $G : \mathbf{Coalg}(\mathbf{D}) \rightarrow \mathbf{Coalg}(\mathbf{C})$ between the corresponding category of models can be derived as shown in [23].

One problem in generalization is the need of type coertions when invoking methods defined in a superclass for subclass object. Methods defined in a superclass as public or protected can be invoked directly in the inheritance hierarchy. If we use the carrier set of a class as the interpretation of its type, then the carrier set of the subclass is the subset of that of its superclass. Therefore, we can use an injection map between them to describe the reuse of a method defined in a superclass (U_p) as the following diagram shows:

$$\begin{array}{ccc} U_p \times B & \xrightarrow{m_p} & U_p \times C \\ {\scriptstyle \iota \times id}\uparrow & & \uparrow{\scriptstyle \iota \times id} \\ U_s \times B & \xrightarrow{m_s} & U_s \times C \end{array}$$

If a method in a superclass is overloaded in its subclass, we only need to define it as a new method in the subclass with the same name as the method in the superclass being overloaded and appropriate

type of parameters and result. In fact, overloaded methods are "more germane to implementation than to specification" [12], so for simplicity, we will not discuss overloading in the following since class diagram is mainly used in the analysis and design stages of system development.

Let $(F_{pub}, F_{pro}, F_{pri})$ and $(F'_{pub}, F'_{pro}, F'_{pri})$ be the functor tuples representing the signature of the public, protected and private parts of a superclass $\mathbf{C}$ and its subclass $\mathbf{C}'$ respectively. By definition of the keywords **public**, **protected** and **private** in [46], it is obvious that all F_{pub_i} (F_{pro_i}) can be found as components of F'_{pub} (F'_{pro}) (that means, all the public and protected attributes and methods in the superclass can be found in its subclass, with identical or overloaded definition. Moreover, subclasses may have additional public and protected methods). Consequently, the two projections $p_{pub} : F'_{pub} \rightsquigarrow F_{pub}$ and $p_{pro} : F'_{pro} \rightsquigarrow F_{pro}$ (as natural transformations) exist.

We use $F_{pub_i}|_{\mathbf{C}'}$ and $F_{pro_i}|_{\mathbf{C}'}$ to represent the methods in class $\mathbf{C}'$ corresponding to those in $\mathbf{C}$, then we can get the projections $p_{pub} : F'_{pub} \Rightarrow F_{pub}$ and $p_{pro} : F'_{pro} \Rightarrow F_{pro}$

$$\begin{cases} p_{pub} = \prod\limits_{\prod F_{pub_i} = F_{pub}} (\pi_i : F'_{pub} \to F_{pub_i}|_{\mathbf{C}'}) \\ p_{pro} = \prod\limits_{\prod F_{pro_i} = F_{pro}} (\pi_i : F'_{pro} \to F_{pro_i}|_{\mathbf{C}'}) \end{cases} \tag{5}$$

Now we turn to the inheritance between classes. The following definition gives a special kind of morphism, called *inheritance morphism*, between a class and its superclass.

Definition 4.8 (inheritance morphism) *Suppose class specification* $\mathbf{Spec}'$ *inherits from* $\mathbf{Spec}$*, and consider two coalgebras* c *and* c' *as models of* $\mathbf{Spec}$ *and* $\mathbf{Spec}'$ *respectively. Then an inheritance morphism from* c' *to* c *is defined as a tuple* (G, p_{pub}, p_{pro})*, such that all states in* U' *are mapped by* G *to the states in* U*, and* $G(u'_0) = u_0$*, where* G *is the forgetful functor between the model categories of the two class specifications,* p_{pub} *and* p_{pro} *are the two projections, and the following diagram commutes.*

$$\begin{array}{ccccc} F_{pro}(U) & \xleftarrow{F_{pro}} & U & \xrightarrow{F_{pub}} & F_{pub}(U) \\ {\scriptstyle F_{pro}(G)}\uparrow & & & & \uparrow{\scriptstyle F_{pub}(G)} \\ F_{pro}(U') & & \uparrow G & & F_{pub}(U') \\ {\scriptstyle p_{pro_{U'}}}\uparrow & & & & \uparrow{\scriptstyle p_{pub_{U'}}} \\ F'_{pro}(U') & \xleftarrow[F'_{pro}]{} & U' & \xrightarrow[F'_{pub}]{} & F'_{pub}(U') \end{array}$$

Since a subclass does not inherit the private part of its superclass, we can derive the definition of class morphism which is weaker than the notion of homomorphism of coalgebras.

Definition 4.9 (class morphism) *Suppose* $(U, \alpha : U \to F(U), u_0)$ *and* $(V, \beta : V \to F(V), v_0)$ *are two classes of specification* $\mathbf{Spec}$*, then a function* $f : U \to V$ *is called a class morphism if* f *preserves*

initial state ($f(u_0) = v_0$) and the following diagram commutes:

$$\begin{array}{ccccc} F_{pro}(V) & \xleftarrow{F_{pro}} & V & \xrightarrow{F_{pub}} & F_{pub}(V) \\ {\scriptstyle F_{pro}(f)}\uparrow & & \uparrow{\scriptstyle f} & & \uparrow{\scriptstyle F_{pub}(f)} \\ F_{pro}(U) & \xleftarrow{F_{pro}} & U & \xrightarrow{F_{pub}} & F_{pub}(U) \end{array}$$

With this definition, we can get the following theorem which shows that a model of a subclass specification is a subclass of any model of its superclass specification.

Theorem 4.2 *Let class c and d be two classes of the same specification* $\mathbf{Spec}$*, class c' inherit from class c, the inheritance morphism is (G, p_{pub}, p_{pro}). If there is a class morphism f from c to d, then c' also inherits from class d. The inheritance morphism is $(f \circ G, p_{pub}, p_{pro})$.*

Proof: Immediate from the composition of the two commuting diagrams in Definition 4.8 and 4.9, as the following diagram shows, where the commutability of the whole diagram comes from that of its upper and lower components.

$$\begin{array}{ccccc} F_{pro}(V) & \xleftarrow{F_{pro}} & V & \xrightarrow{F_{pub}} & F_{pub}(V) \\ \uparrow{\scriptstyle F_{pro}(f)} & & \uparrow{\scriptstyle f} & & \uparrow{\scriptstyle F_{pub}(f)} \\ F_{pro}(U) & \xleftarrow{F_{pro}} & U & \xrightarrow{F_{pub}} & F_{pub}(U) \\ \uparrow{\scriptstyle F_{pro}(G)} & & \uparrow & & \uparrow{\scriptstyle F_{pub}(G)} \\ F_{pro}(U') & & {\scriptstyle G} & & F_{pub}(U') \\ \uparrow{\scriptstyle p_{pro}} & & | & & \uparrow{\scriptstyle p_{pub}} \\ F'_{pro}(U') & \xleftarrow[F'_{pro}]{} & U' & \xrightarrow[F'_{pub}]{} & F'_{pub}(U') \end{array}$$

In this framework, the semantics of the generalization relationship in UML class diagrams can be given as all the possible inheritance morphisms between the models of the corresponding class specifications.

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

where g is the inheritance morphism from d to c.

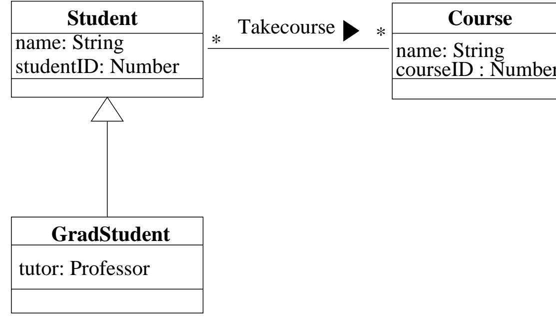


Figure 13: Association being inherited

Substitutability is an important property of generalization. It asserts that any instance of a subclass can be used wherever objects of the superclass are expected without changing the behavior pattern. More precisely, the subclass can simulate the behavior of its superclass. The pair $p = (p_{pub}, p_{pro})$ between the signature functors of the two classes which is defined by projections ensures that the methods' signature of the superclass are consistent with that of the subclass.

Theorem 4.3 *Let $h = (g, p)$ be an inheritance morphism between class* $\mathbf{C}'$ *and* $\mathbf{C}$*, F' is a method functor in class* $\mathbf{C}'$ *being inherited from class* $\mathbf{C}$*, then for all objects o of* $\mathbf{C}'$*, $p(F')(g(o)) = h(F'(o))$.*

Proof: Consider the diagram given in the proof of Theorem 4.2. Suppose $\forall u' \in U'$ be the state of o, then

$$
\begin{aligned}
h(F'(o)) =& h(F'(u')) \\
=& (p(F')(f \circ G))(p(F')(u')) \\
=& p(F')((f \circ G)(u')) \\
=& p(F')(g(u')) \\
=& p(F')(g(o))
\end{aligned}
$$

Associations are inherited by subclasses. See Figure 13 as an example. An association of class **Student** must be applicable when objects of class **Student** are substituted by objects of class **GradStudent**. The representation of association **Takecourse** has been given by a pair of coalgebraic functions $student$ and $course$, here we define another pair of functions to represent the inherited association between class **GradStudent** and **Course**:

$$
\begin{aligned}
& student_{gs} : Course \to \mathcal{P}(GradStudent), \\
& course_{gs} : GradStudent \to \mathcal{P}(Course)
\end{aligned}
$$

The semantics of the functions above are constrained by the functions of the corresponding association

of the superclass $Student$:

$$student_{gs}(c) = student(c)|_{\mathsf{GradStudent}}$$
$$course_{gs}(gs) = course(g(gs))$$

Here $student(c)|_{\mathsf{GradStudent}}$ means the restriction of the set of students taking course c to the subset in which all the elements are graduate students taking this course, and g is the inheritance morphism between class $\mathbf{Student}$ and $\mathbf{GradStudent}$.

The semantic function is given as:

$$\begin{aligned} \mathcal{S}[\![\mathbf{B}_m^b \text{—}_n^c \mathbf{C} \;\lhd\!\!\text{—}\; \mathbf{D}]\!] \triangleq & \{((b,c), g, (b',d)) \mid (b,c) \in \mathcal{S}[\![\mathbf{B}_m^b \text{—}_n^c \mathbf{C}]\!] \wedge \\ & g \in \mathcal{S}[\![\mathbf{C} \;\lhd\!\!\text{—}\; \mathbf{D}]\!] \wedge (b',d) \in \mathcal{S}[\![\mathbf{B}_{m'}^{b'} \text{—}_{n'}^d \mathbf{D}]\!] \wedge \\ & m' \subseteq m \wedge (\forall o_U = (id,u) : \mathbf{D}.b'(o_U) = b((id, g(u)))) \wedge \\ & (\forall o_V : \mathbf{B}, o_U = (id,u) : \mathbf{D}.o_U \in d(o_V) \Rightarrow \\ & (id, g(u)) \in c(o_V))\} \end{aligned}$$

where g is the corresponding inheritance morphism for the generalization between $\mathbf{C}$ and $\mathbf{D}$. The predicate $m' \subseteq m$ specifies the restriction on the number of objects of class $\mathbf{B}$ being linked to an object of class $\mathbf{D}$ and class $\mathbf{C}$ respectively. Furthermore, the substitutability property is also satisfied.

4.3.1 Abstract Classes

Generalization relationships organize classes in 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 r espectively. An abstract class may not have direct instances. Therefore, we can not interpret it in the same way as concrete classes. However, from the generalization relationship between an abstract class and its subclasses, we can get its semantics as the smallest superclass of all 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 subclass coalgebras.

Definition 4.10 (semantics of abstract class) *The semantics of an abstract class is the colimit of its subclasses in the category of classes with the inheritance morphisms as arrows for the colimit.*

$$\mathcal{S}[\![\mathbf{C}\{Abstract\} \;\lhd\!\!\text{—}\; {}^*\{\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C_n}\}]\!] \triangleq \mathbf{Colimit_{Coalg}}(c_1, c_2, \ldots, c_n)$$

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

An abstract class may have several subclasses and can only be instantiated by one of them. (such as class **Person** (P) in Figure 14). If the direct subclasses form a partition of the abstract class, that means,

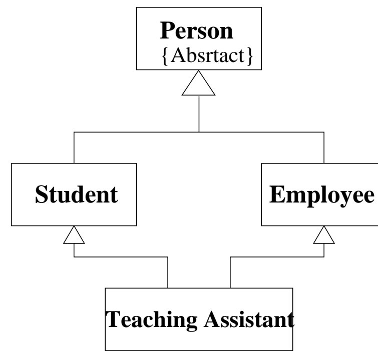
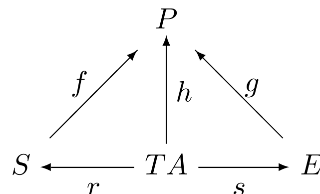


Figure 14: Multiple inheritance with common ancestor

the subclasses have no common subclass, then we can use a coproduct of the subclasses in the category of classes to represent the abstract class and the coprojections of the coproduct are the inheritance morphisms. Otherwise, the subclasses of a common abstract class may be not disjoint, such as in Figure 14, class **Student** (S) and **Employee** (E) have a common subclass **TeachingAssistant** (TA). In this situation, the interpretation can be generalized to the colimit of the subclasses. The arrows describing the inheritance relationships in this example are: $f : S \to P$, $g : E \to P$, $r : TA \to S$, $s : TA \to E$, then P is the colimit of the subclasses S, E, and TA, the corresponding colimit arrows are f, g, h, where $h = f \circ r = g \circ s$. It is natural that the commuting property of colimit holds, as the following diagram shows:



Two less common but useful notions are *isRoot* and *isLeaf* which specifies whether a class may have no parents or children. Especially when we have multiple, independent inheritance lattices, it is useful to designate the top and bottom of each hierarchy via the root and leaf classes.

The semantics of root and leaf classes are given as follows:

$$\mathcal{S}[\![\mathbf{C}\{root\}]\!] \triangleq \mathcal{S}[\![\mathbf{C}]\!] \wedge (\forall \mathbf{D}.\mathcal{S}[\![\mathbf{D} \ \lhd\!\!\!\longleftarrow \mathbf{C}]\!] = \{\ \})$$
$$\mathcal{S}[\![\mathbf{C}\{leaf\}]\!] \triangleq \mathcal{S}[\![\mathbf{C}]\!] \wedge (\forall \mathbf{D}.\mathcal{S}[\![\mathbf{C} \ \lhd\!\!\!\longleftarrow \mathbf{D}]\!] = \{\ \})$$

which are same as semantics of other classes, plus the condition that no superclass of a root class **C** (and subclass for leaf class, respectively) is permitted to appear in the class diagram.[6]

[6]Here we abuse the operator $\wedge$. The precise meaning of $\mathcal{S}[\![\mathbf{C}]\!] \wedge p$ is a subcategory of $\mathcal{S}[\![\mathbf{C}]\!]$ where all objects in the subcategory satisfy p.

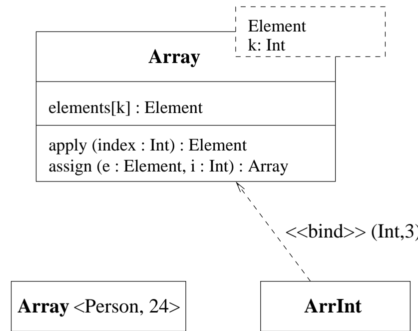


Figure 15: Template classes and instantiated classes

4.3.2 Multiple Inheritance

A class may have more than one parent in a UML class diagram, a fact known as *multiple inheritance*. It means that the subclass has all the attributes, methods and associations of all the superclasses. We assume that the multiple inheritance being concerned is well-formed, that is, an attribute or method with the same signature can not be declared independently by several superclasses that do not inherit it from a common ancestor. Then the semantics for a class that multiplicitly inherits from superclasses are their lower bound coalgebras. Thus, the semantics of a class $\mathbf{C}$ multiply inherits from a set of classes $\{\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_n\}$, is given by the cones of $c_1, c_2, \ldots, c_n$ where c_i are the coalgebras in $\mathcal{S}[\![\mathbf{C}_i]\!]$ respectively, in which the arrows for the cone are the corresponding inheritance morphism.

$$\mathcal{S}[\![\mathbf{C} \longrightarrow\!\!\triangleright \{\mathbf{C}_1, \mathbf{C}_2 \ldots, \mathbf{C}_n\}]\!] \triangleq \mathbf{Cone_{Coalg}}(c_1, c_2, \ldots, c_n)$$

In the semantics of multiple inheritance, we can get the limit d of the superclass set $\{c_1, c_2, \ldots, c_n\}$ which is the terminal object in the category of all the possible cones of $\{c_1, c_2, \ldots, c_n\}$, that means, d has no more attributes or methods than those appears in $\{c_i\}_{i=1,\ldots,n}$, and for all $c \in Obj\mathcal{S}[\![\mathbf{C}]\!]$, c is a subclass of d.

4.4 Templates

UML allows the use of *templates* or *parameterized classes*. A template has one or more unbound formal parameters. It describes a family of classes, each of which is specified by binding the parameters to actual values.

A template can not be used directly because it has a free parameter that is not meaningful outside of a scope that declares the parameter. A template must be instantiated first before it is used, i.e. the template's parameters must be bound to actual values.

As shown in Figure 15, a template can be instantiated in two ways: Either implicitly by declaring a class whose name provides the binding or explicitly using a *bind* relationship specifying the source instantiates the target template using the actual parameters.

Since a template can not have instances of itself, it can only have one direction associations from itself to other classes. Moreover, it can not be a superclass of other classes, but its instances can.

Let $\mathbf{C}{:}P{:}$ be a template class where P is the list of parameters. Then

$$\mathcal{S}[\![\mathbf{C}{:}P{:}\,{}^{c}\!\!-\!\!-{}^{d}\mathbf{D}]\!] \triangleq False$$

$$\mathcal{S}[\![\mathbf{C}{:}P{:}\,{}^{c}\!\!\longleftarrow{}^{d}\mathbf{D}]\!] \triangleq False$$

$$\mathcal{S}[\![\mathbf{C}{:}P{:}\;\lhd\!\!\!\longleftarrow \mathbf{D}]\!] \triangleq False$$

Here the **Bool** value $False$ is used to denote the empty set { }, i.e., the diagram can not be implemented.

An instantiation of a template is a class specification which is decided by one element in $\mathcal{S}[\![P]\!]$. Its semantics is:

$$\mathcal{S}[\![\mathbf{C} < p >]\!] \triangleq \begin{cases} p \in \mathcal{S}[\![P]\!] \Rightarrow \mathcal{S}[\![\mathbf{C}[p/P]]\!] \\ else \Rightarrow False \end{cases}$$

which also gives the semantics of binding relationship in a class diagram.

4.5 Semantics of Class Diagrams

Up to now, the discussion has been on the class level including the relationship between classes. Finally, the semantics of class diagrams can be given. A class diagram represents a system composed by objects of specified classes. Its semantics can be defined coalgebraically via object diagrams.

An object diagram is a snapshot of a corresponding class diagram. It exhibits a set of objects and their relationships existing in a system at a point of time. An example of an object diagram is given in Figure 16. The objects and their relationships in an object diagram are defined by the semantics 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. It can be used to describe the existence of certain objects together with the relationships between them in the system at a certain time. Therefore, an element $\sigma \in \Sigma$ is interpreted as the product of states of different objects at the same point of time. Consequently, the semantics of an object diagram is an element $\sigma \in \Sigma$ which describes a state of the system, and the system being modeled can be described as a coalgebra $(\Sigma, c : \Sigma \to F(\Sigma))$, where c describes all the possible transitions between system states.

Now we can get the following definition of class diagram semantics:

Definition 4.11 (class diagram semantics) *The semantics of a class diagram* **CD** *is defined as a cate-*

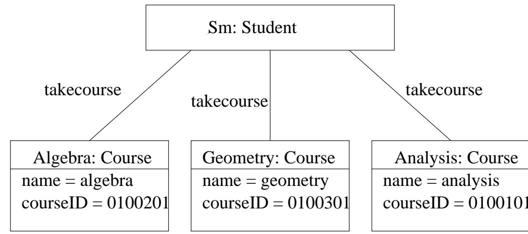


Figure 16: Object diagram

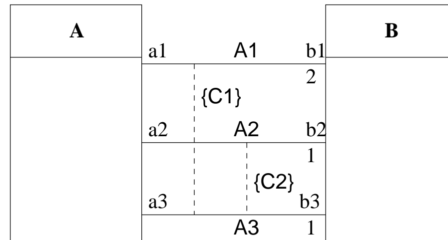


Figure 17: Inconsistency of a class diagram

gory denoted as **Coalg(CD)***. The objects in this category are the coalgebras* $(\Sigma, c : \Sigma \to F(\Sigma))$ *where* Σ *is the system state space of all the possible system states.* F *is the tensor product composed by the signature functors of the component classes and associations in the class diagram, which describes the possible system state transitions and observations to the system states. The arrows are* F*-homomorphisms between them.*

Remark. Generally, the state space of a system may be different at different time. For example, objects may be created or deleted. Thus, the system may be initiated in a state σ in the state space Σ, but we permit operations of the system that change the state space, so that the system state may be in another state space Γ later. However, in this situation, we can take $\Sigma \cup \Gamma$ as the system state space. Then the change of state space, like creating new objects or deleting old objects, also becomes a state transition.

4.6 Examples in Checking Consistency of Class Diagrams

In this section we use some simple examples to illustrate the approach for checking the consistency of class diagrams via the coalgebraic semantics defined previously.

Take Figure 17 as an example [7], where C1 is a constraint which states that every link in A1 is either in A2 or in A3. The constraint C2 states that A2 and A3 are equal. This diagram seems to be consistent,

[7]This example is taken from [3].

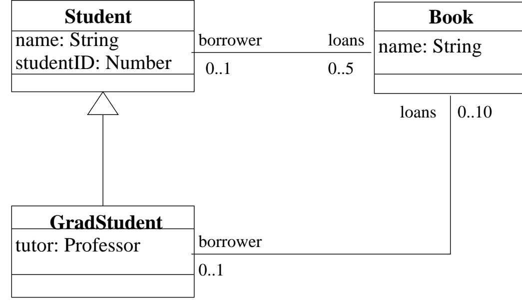


Figure 18: An inconsistent class diagram for a library system

but it is not. We first give the semantics of the two constraints explicitly as follows:

$$\mathcal{S}[\![\mathsf{C1}]\!] \triangleq \forall u : A, v : B.(b1(u) \subseteq b2(u) \cup b3(u)) \wedge a1(v) \subseteq a2(v) \cup a3(v))$$

$$\mathcal{S}[\![\mathsf{C2}]\!] \triangleq \forall u : A, v : B.(u \in a2(v) \Leftrightarrow u \in a3(v)) \wedge (v \in b2(u) \Leftrightarrow v \in b3(u))$$

For an object u of class $\mathbf{A}$, from the semantics of $\mathsf{C1}$ and $\mathsf{C2}$ we have

$$\begin{aligned}
&\mathcal{S}[\![\mathsf{C1}]\!] \wedge \mathcal{S}[\![\mathsf{C2}]\!] \\
=&\forall u : A, v : B.(b1(u) \subseteq b2(u) \cup b3(u)) \wedge a1(v) \subseteq a2(v) \cup a3(v) \wedge a2(v) = a3(v) \wedge b2(u) = b3(u)) \\
=&\forall u : A, v : B.(b1(u) \subseteq b2(u) = b3(u) \wedge a1(v) \subseteq a2(v) = a3(v)) \\
\Rightarrow&\forall u : A, v : B.(card(b1(u)) \leq card(b2(u)) = card(b3(u)) \wedge card(a1(v)) \leq card(a2(v)) = card(a3(v)))
\end{aligned}$$

From the semantics of associations, we have

$$\forall u : A, v : B.card(b1(u)) = 2 \wedge card(b2(u)) = 1 \wedge card(b3(u)) = 1$$

Therefore, we have

$$\mathcal{S}[\![A1]\!] \wedge \mathcal{S}[\![A2]\!] \wedge \mathcal{S}[\![A3]\!] \wedge \mathcal{S}[\![\mathsf{C1}]\!] \wedge \mathcal{S}[\![\mathsf{C2}]\!] \Rightarrow 2 \leq 1 \equiv False$$

So the diagram is inconsistent. It can not be implemented correctly.

Now take Figure 18 as another example for a library system. There are two classes $\mathbf{Student}$ and $\mathbf{Book}$ and an association which shows the relationship between students and books. Every student can borrow at most 5 books at the same time. Later another class $\mathbf{GradStudent}$ is added to the diagram which is desired to be the subclass of class $\mathbf{Student}$. And every graduate student can borrow at most 10 books at the same time. From the semantics definition for inheritance of associations, we can easily derive that

$$\mathcal{S}[\![\mathbf{Book}^{loans}_{0..5} \text{---} ^{borrower}_{0..1}\mathbf{Student} \ \triangleleft\!\!\text{---} \ \mathbf{GradStudent}]\!] \Rightarrow 0..10 \subseteq 0..5 \Rightarrow False$$

Therefore the diagram is inconsistent. To solve the problem, we can define class $\mathbf{Student}$ as an abstract class and add $\mathbf{UnderGraduate}$ as another subclass of class $\mathbf{Student}$, and then change the association between $\mathbf{Book}$ and $\mathbf{Student}$ in Figure 18 to an association between this subclass and $\mathbf{Book}$, as Figure 19 shows. Moreover, because every book can be borrowed by at most one student at any time point, a constraint $\{xor\}$ is added to state that the two associations can not exist at the same time for one object of class $\mathbf{Book}$.

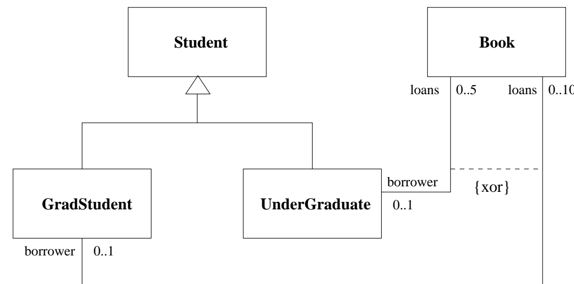


Figure 19: A consistent class diagram

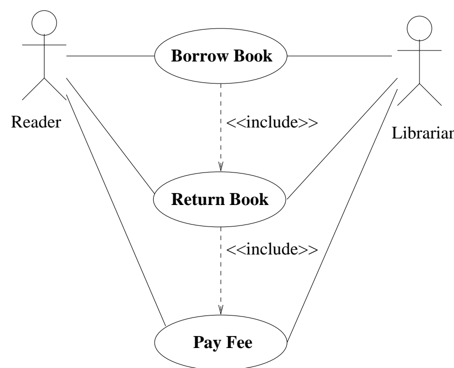


Figure 20: Use Case Diagram

5 Use Cases

Use Cases, first introduced by Jacobson [33, 34] and widely used in object-oriented analysis and design methods, are popular models for capturing requirements, primarily functional requirements and business requirements [55]. They help to identify the primary requirements of the system and form the units of work in an incremental, use case driven, object-oriented software development process like the Rational Unified Process [21, 37] which is relatively widely used.

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" [33]. Use cases are very useful in decomposing and capturing the requirements, gaining an understanding of the problem domain and identifying the different goals for individual actors and the system. Use cases are specified informally, which makes them successful in capturing requirement.

Use cases are text documents, not diagrams. However, the UML defines the use case diagram to illustrate the name of use cases, actors and the relationships among them. See Figure 20 as an example of use case diagrams where the **Borrow Book** use case includes the **Return Book** use case, which includes another **Pay Fee** use case. These use cases are written in Figure 21, 22 and 23 as flows of events with pre- and

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.

Figure 21: Borrow Book use case

Use Case	Return Book
Actors	Reader, Librarian
Precondition	The reader has borrowed the book and it is not returned back by the reader.
Flow of Events	1. The use case begins when the reader chooses a book to return;
	2. The book is returned and the reader is no more the borrower of the book;
	3. The librarian checks whether the reader should pay for his/her debts;
	4. If the reader should pay fine for debts Include Pay Fee.
Postcondition	The reader has successfully returned the book.

Figure 22: Return Book use case

post-conditions together indicating what states the system must be in at the beginning and end of a use case.

Such use cases can be considered as contracts between the system and actors specifying the conditions that the behaviors of the system should satisfy, which may be formalized by coalgebraic specifications [29]. A use case can be interpreted coalgebraically as a sequence of actions followed by some observations. To measure the effect of the actions we relate the observations in a coalgebraic definition. This means that use cases can be defined coinductively. Then single actions represent *atomic use cases*, which change the system from one state to another.

The signature of an atomic use case $\mathbf{UC}$ is defined by a functor F. Let Σ be the system state space, as defined in the semantics of class diagrams. Then a use case is interpreted as a coalgebraic partial function of type $uc : \Sigma \to F(\Sigma)$. This function is meaningful when the possible system states before the action satisfy the given pre-condition. An example should demonstrate the coinductive formalization of use cases.

Consider a use case $\mathbf{BB}$ of borrowing a book in our previous library example (Figure 18). Let Σ be the state space of the system. Then the behavior of $\mathbf{BB}$ can be defined by a coalgebraic function $bb : \Sigma \to$

Use Case	Pay Fee
Actors	Reader, Librarian
Precondition	The reader has a debt to the system;
Flow of Events	1. The use case begins when the reader has a debt to the system;
	2. The reader chooses an amount of the debt and pays it;
	3. The librarian subtracts the sum from the reader's debt.
Postcondition	The reader has successfully payed the debt.

Figure 23: Pay Fee use case

$\Sigma^{Reader \times Book}$. If a reader s_0 borrows book b_0, then the observable effects can be specified coinductively:

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

where s_0', b_0' are observers on Σ for obtaining the objects s_0, b_0 and $\sigma \in \Sigma$. Here, the precondition $\mathbf{pre}$ guarantees the multiplicity constraints in the class diagram.

To express the **Borrow Book** use case model completely, in the following, the RSL specifications for the use cases are given. First, two schemes are defined for the classes involved as shown in Figure 24 and 25, which are essentially needed within our system. Then the specification for the use cases are defined as in Figure 26, which describes all the three use cases by the axioms.

5.1 Discussions on Advanced Techniques

When the flow of events is refined, we may find similarity in the various use cases that we want to abstract into a common place, extend a use case without changing the original description, or find a lot of similarities in some of the actors. To take the advantage of these similarities in the system, some techniques may be applied.

We may have something generic that can be reused. The common behavior can be abstracted with an *include* relationship. Defining an included use case Starts by identifying the steps that we want to use in many places and put the steps in a use case and give them a name. A use case can include any number of other use cases. We can have as many levels of including as we desire. An include relationship of use cases means that the base use case incorporate the behavior of the target use cases. The behaviors in the target use case is included at one point of the base use case.

A use case can be specialized into one or more child use cases, each of the child use cases contains all the observations and behaviors in the super use case, and may add more behaviors into the behavior sequences. This is a use case *generalization* relationship.

An *extend* relationship means that a use case can be defined as an incremental extension to another use case. The base use case may not depend on the extending use case. The extend relationship contains

```
scheme BOOK =
  class
    type
      Book,
      Person,
      Date = Nat,
      Status == free | lent
    value
      status : Book → Status,
      returnDate : Book ~→ Date,
      borrower : Book ~→ Person,
      book : Book,
      setLoan : Book × Person × Date ~→ Book,
      resetLoan : Book ~→ Book,
      today : Date,
      payFee : Book → Int
    axiom
      status(book) ≡ free,
      ∀ b : Book, p : Person, d : Date •
        status(setLoan(b,p,d)) ≡ lent
        pre status(b)=free,
      ∀ b : Book, p : Person, d : Date •
        borrower(setLoan(b,p,d)) ≡ p
        pre status(b)=free,
      ∀ b : Book, p : Person, d : Date •
        returnDate(setLoan(b,p,d)) ≡ d + 30
        pre status(b)=free,
      ∀ b : Book •
        status(resetLoan(b)) ≡ free
        pre status(b)=lent,
      ∀ b : Book •
        payFee(b) ≡
          let rd = returnDate(b) in
            if rd < today then 0
            else rd − today
            end
          end
  end
```

Figure 24: RSL specification for BOOK

```
context: BOOK
scheme READER =
  class
    object
      B : BOOK
    type
      Reader,
      Book = B.Book,
      Loan = Book-set
    value
      fine : Reader → Int,
      borrowBook : Reader × Book →̃ Reader,
      returnBook : Reader × Book →̃ Reader,
      loans : Reader → Loan,
      payFine : Reader × Int →̃ Reader,
      reader : Reader
    axiom
      fine(reader) ≡ 0,
      loans(reader) ≡ {},
      ∀ r : Reader, b : Book •
        loans(borrowBook(r,b)) ≡ loans(r) ∪ {b}
        pre B.status(b)=B.free,
      ∀ r : Reader, b : Book •
        fine(borrowBook(r,b)) ≡ fine(r)
        pre B.status(b)=B.free,
      ∀ r : Reader, b : Book •
        loans(returnBook(r,b)) ≡ loans(r) \ {b}
        pre b ∈ loans(r),
      ∀ r : Reader, b : Book •
        fine(returnBook(r,b)) ≡ fine(r) + B.payFee(b)
        pre b ∈ loans(r),
      ∀ r : Reader, i : Int •
        fine(payFine(r,i)) ≡
          if i<fine(r) then fine(r)−i
          else 0
          end
      pre fine(r) ≥ i,
      ∀ r : Reader, i : Int •
        loans(payFine(r,i)) ≡ loans(r)
  end
```

Figure 25: RSL specification for READER

```
context: BOOK,READER
scheme LIBRARY =
  class
    object
      R : READER
    type
      Book = R.Book,
      Reader = R.Reader
    channel
      reader : Reader,
      book : Book,
      fee : Int
    value
      borrowBook : Unit → in reader,book out reader,book Unit,  --The Borrow Book Use Case
      returnBook : Unit → in reader,book out reader,book Unit,  --The Return Book Use Case
      payFee : Unit → in reader,fee out reader Unit             --The Pay Fee Use Case
    axiom
      ∀ b1 : R.Book •
          borrowBook() ≡
            let (b,r)=(book?,reader?) in
              if R.B.status(b)=R.B.free then
                if (card(R.loans(r))=R.limit ∧ b1 ∈ R.loans(r)) then
                  reader!(R.return(r,b1));
                  book!(R.B.resetLoan(b1));
                  if R.fine(r)>0 then
                    reader!(R.payFine(r,R.fine(r)))
                  end
                end;
                book!(R.B.setLoan(b,r,R.B.today));
                reader!(R.borrow(r,b))
              end
            end,
          returnBook() ≡
            let (b,r)=(book?,reader?) in
              if (R.B.status(b)=R.B.lent ∧ R.B.borrower(b)=r) then
                reader!(R.return(r,b));
                book!(R.B.resetLoan(b));
                if R.fine(r)>0 then
                  reader!(R.payFine(r,R.fine(r)))
                end
              end
            end,
          payFee() ≡
            let(f,r)=(fee?,reader?) in
              reader!(R.payFine(r,f))
            end
  end
```

Figure 26: RSL specification for the use cases

a condition which should be satisfied if the extension is to happen and a sequence of extension points which defines the location in the base use case where the extension is to take place. This is the same situation as the generalization relationship of use cases, but the extending use case is inserted into the base use case at the extension point conditionally, while in the generalization relationship, the parent use case is replaced by the child use case conditionally.

A family of operations has been defined in [60], including sequential composition ;, choice $+$, parallel composition $\boxtimes$ and $\otimes$, etc., which can be used to build more complex use cases from atomic use cases. So that the semantics for the relationships among use cases like *include, extend* associations and generalizations of use cases can be given by these operations. For example, the use case **BB** is defined as a sequential of the atomic actions corresponding to events $1, 2$, another use case for returning book corresponding to event 3 when the condition is $true$ and event 4.

6 Related Work

A lot of works on specifying the formal semantics of UML can be found in the literature (see e.g. [6, 7, 11, 12, 13, 14, 19, 22]). However, most of them do not give a semantics for the whole language, but only concentrate on part of it, such as class diagrams or statechart diagrams, and there is a lack of research on checking the consistency between different UML models, which is a central issue in multi-view modeling languages such as UML.

The meta-model approach adopted by the UML Semantics [45] use a UML class diagram together with constraints as a meta model and define the semantics of the language via it. However, a problem in this approach is that it defines the semantics of elements in terms of those elements whose semantics are not precisely defined. In fact, the well-formedness rules are descriptions at the syntactic level, but not the semantics definition.

From a logical perspective, a set of models are consistent means that they are free from contradictions. In other words, they have a common semantic interpretation. So a natural idea is to define an independent semantic domain in which all models can be interpreted. [6] sketches a general scenario for several UML diagrams but no technical details are provided. Recently, some researchers also try to combine the formalization of different UML models, see e.g. [18, 42].

An alternative approach is to translate UML into another formal language that is well understood. In [7] a translation from UML models to Z specifications is provided. [58] studies UML class diagrams by means of using Object-Z. [49] discusses active classes. [15] provides a translation from UML class diagrams to RSL [63] specifications. A translation from UML class diagrams to O-Slang is provided in [57]. Unfortunately, most of them also only focus on part of UML models.

The inspiration for our own coalgebraic semantic domain came from the work of Jacobs on object-oriented systems [24, 25, 29] and that of Tews on the relationship between CCSL and UML [62]. We use their concept of coalgebraic specifications to give UML diagrams a semantics. However, in contrast to their work on the coalgebraic specification language CCSL we take a rather pragmatic approach and

aim to use UML diagrams to denote coalgebraic specifications. A further difference to previous work and to very recent work on UML class diagram semantics [38, 41] is that we take a categorical rather than a set-theoretic approach. The motivation is to be more general and flexible and, thus to be able to cover a broader subset of UML. Already, the presented work on class diagrams shows the advantages of having categorical tools, like morphism, functors, natural transformations etc., available. One advantage of the cofibred approach is that it provides us with a single category which contains models (coalgebras) corresponding to different specifications (UML classes) and allows us to relate coalgebras corresponding to different specifications within one category. Another benefit of the categorical approach is that we can construct the final coalgebra in the semantics for a given UML class which can be used as the minimal implementation as shown in [59]. Our future work on the dynamic aspects of UML, like statechart diagrams, will make these benefits even more obvious.

In UML statechart diagrams describe the dynamic behaviour of systems. Several formal semantics for statechart diagrams have been proposed previously. For example, in [16] input-output labeled transition systems (LTS) are used as the semantic domain. From these previous results it can be deduced that statecharts diagrams can be straightforwardly interpreted in a coalgebraic framework. For examples of LTSs defined as coalgebras we refer to [54]. Thus, the semantics of a statechart diagram will be a coalgebra. Then, consistency checks between class diagrams and statechart diagrams are 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.

Compared to others' approach, directly defining the coalgebraic semantics can make us model the system more faithfully since we have the freedom to choose the signature functor appropriately. Concerning verification of system, the coalgebraic approach often allows for a smaller state space of the model than encoding the system as a Kripke structure and make the verification time of the system property shorter (as shown by D. Pattinson in [47]). Furthermore, by resorting the underlying environment category from $\mathbf{Set}$ to a set based category enriched with some algebraic structure, we naturally get a data refinement of the state space representation.

It is already evident from our ongoing work that we will be able to stay in the domain of coalgebras for defining semantics of other UML diagrams, like statecharts and interaction diagrams. Generally, it can be said that our attempt of having one semantic domain for different UML diagrams contrasts us from most of the previous works on this topic.

7 Conclusion and Discussion

In this paper, essential parts of a coalgebraic semantics for UML class diagrams have been presented. It has been shown in detail how Classes, their Associations and Generalizations in a UML diagram can be interpreted as coalgebraic specifications. Furthermore an outlook on the formalization of use-case diagrams (and statechart diagrams) has been given. Although, no more technical discussion of other diagrams is provided, the reader should by now get an idea how a coalgebraic semantics facilitate the integration of static and dynamic aspects of UML.

This paper is only a starting point for the formalization of UML. The next steps include to define the coalgebraic semantics of other UML models, such as statechart diagrams, to give a precise meaning for the consistency of models, to do research on the refinement of different kinds of models which involves a transformation of functors for different steps. However, having a formal semantics is not enough. The next step will involve research on applications of the semantics: A use-case driven development method for UML diagrams, supported by algebraic laws and coalgebraic proof methods, will be designed. Based on the semantics and method, tools are planned, including a model checker and a test-case generator for UML diagrams. Moreover, the work of Jacobs and Poll on coalgebraic semantics of Java [30] shows the possiblity of a semantic-based approach for code generation from UML models.

Currently, the opinions on UML in the research community are twofold. Some see the disadvantages of UML and reject the language completely. Others think that it is our duty to improve this de-facto standard and help software engineers in the application of UML. We belong to the second group and hope that our work will contribute to the improvement of UML and its associated methods.

Acknowledgments

We would like to thank Prof. Zhang Naixiao and Luís Barbosa for many detailed comments and continuous advices on the draft of this paper. This work is partially supported by the National Natural Science Foundation of China under Grant No. 60273001.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] P. Aczel. *Non-well-founded sets*. Number 14 in CSLI Lecture Notes. Center for the Study of Languages and Information, Stanford, 1988.

[3] P. André, A. Romanczuk-Réquilé, J.-C. Royer, and A. Vasconcelos. Checking the Consistency of UML Class Diagrams using Larch Prover. In T. Clark, editor, *ROOM 3 (the third Rigorous Object-Oriented Methods Workshop), Proceedings*. BCS eWics, 2000.

[4] M. Barr and C. Wells. *Category Theory for Computing Science, Third Edition*. Les Publications CRM, 1999.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[6] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *LNCS*, pages 344–366. Springer-Verlag, 1997.

[7] J.-M. Bruel and R. B. France. Transforming UML Models to Formal Specifications. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*. Springer, 1999.

[8] J.-M. Bruel, B. Henderson-Sellers, F. Barbier, A. L. Parc, and R. B. France. Improving the UML metamodel to rigorously specify aggregation and composition. In S. Patel, Y. Wang, and R. H. Johnston, editors, *OOIS, 7th International Conference on Object Oriented Information Systems*, pages 5–14. Springer, 2001.

[9] C. Cîrstea. An algebra-coalgebra framework for system specification. In H. Reichel, editor, *Electronic Notes in Theoretical Computer Science*, volume 33. Elsevier Science Publishers, 2000.

[10] C. Cîrstea. *Integrating Observations and Computations in the Specification of State-Based Dynamical Systems*. PhD thesis, Corpus Christi College and St. John's College, University of Oxford, 2000.

[11] T. Clark and A. Evans. Foundations of the unified modeling language. In *2nd Northern Formal Methods Workshop, Ilkley, electronic Workshops in Computing*. Springer-Verlag, 1998.

[12] S. DeLoach and T. C. Hartrum. A Theory-Based Representation for Object-Oriented Domain Models. *Software Engineering*, 26(6):500–517, 2000.

[13] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 336–348. Springer, 1999.

[14] L. Favre and S. Clérici. Integrating UML and Algebraic Specification Techniques. In C. Mingins, editor, *Proceedings of TOOLS Pacific 1999*. IEEE Computer Society, 1999.

[15] A. Funes and C. George. Formal Foundations in RSL for UML Class Diagrams. Technical Report 253, UNU/IIST, May 2002.

[16] S. Gnesi, D. Latella, and M. Massink. Modular Semantics for a UML Statechart Diagrams kernel and its extension to Multicharts and Branching Time Model Checking. *The Journal of Logic and Algebraic Programming*, 51(1):43–75, 2002.

[17] M. Gogolla and M. Richters. Expressing UML Class Diagrams Properties with OCL. In T. Clark and J. Warmer, editors, *Object Modeling with OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*, pages 85–114. Springer, 2002.

[18] M. Große-Rhode. Formal concepts for an integrated internal model of the UML. In H. Ehrig, C. Ermel, and J. Padberg, editors, *Proc. Uniform Approaches to Graphical Process Specification Techniques (UNIGRA) at ETAPS 2001*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[19] A. Hamie, J. Howse, and S. Kent. Modular Semantics for Object-Oriented Models. In *Proceedings of Northern Formal Methods Workshop*, eWics Series. Springer-Verlag, 1998.

[20] B. Henderson-Sellers and F. Barbier. Black and White Diamonds. In R. B. France and B. Rumpe, editors, *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *LNCS*, pages 550–565. Springer, 1999.

[21] J. Hunt. *The Unified Process for Practitioners: Object Oriented Design, UML and Java*. Practitioner. Springer, 2001.

[22] H. Hussmann, M. Cerioli, G. Reggio, and F. Tort. Abstract Data Types and UML Models. Technical Report DISI-TR-99-15, DISI – Università di Genova, Italy, 1999.

[23] B. 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.

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

[25] B. Jacobs. Coalgebraic reasoning about classes in object-oriented languages. In *Electronic Notes in Theoretical Computer Science*, volume 11. Elsevier Science Publishers, 1998.

[26] B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, 1999.

[27] B. Jacobs. The Temporal Logic of Coalgebras via Galois Algebras. Technical Report CSI-R9906, Computer Science Institute, University of Nijmegen, April 1999.

[28] B. Jacobs. Object-oriented hybrid systems of coalgebras plus monoid actions. *Theoretical Computer Science*, 239:41–95, 2000.

[29] B. 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.

[30] B. Jacobs and E. Poll. Coalgebras and monads in the semantics of Java. *Theoretical Computer Science*, 291:329–349, 2003.

[31] 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.

[32] B. Jacobs and H. Tews. Assertional and Behavioural Refinement in Coalgebraic Specification. In *Electronic Notes in Theoretical Computer Science*, volume 47. Elsevier Science Publishers, 2001.

[33] I. Jacobson. Object Oriented Development in an Industrial Environment. In N. 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.

[34] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[35] A. Kurz. *Logics for Coalgebras and Applications to Computer Science*. PhD thesis, Universität München, 2000.

[36] A. Kurz. Specifying coalgebras with modal logic. *Theoretical Computer Science*, 260:119–138, 2001.

[37] C. Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, 2002.

[38] X. Li, Z. Liu, and J. He. Formal and Use-Case Driven Requirement Analysis in UML. Technical Report 230, UNU/IIST, March 2001.

[39] Z. Liu. Object-Oriented Software Development Using UML. Technical Report 229, UNU/IIST, March 2001.

[40] Z. Liu. Software Development with UML. Technical Report 259, UNU/IIST, July 2002.

[41] Z. Liu, J. He, and X. Li. Formalizing the Use of UML in Requirement Analysis. Technical Report 228, UNU/IIST, March 2001.

[42] Z. Liu, X. Li, and J. He. Using Transition Systems to Unify UML Requirement Models. Technical Report 263, UNU/IIST, October 2002.

[43] B. Meyer. *Object-oriented Software Construction (2nd edition)*. Prentice Hall, 1997.

[44] L. Moss. Coalgebraic logic. *Annals of Pure and Applied Logic*, 96:277–317, 1999.

[45] OMG. *OMG Unified Modeling Language Specification, Version 1.3* , 2000.

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

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

[48] E. Poll. A coalgebraic semantics of subtyping. In H. Reichel, editor, *Proceedings 3rd Workshop on Coalgebraic Methods in Computer Science, CMCS'2000, Berlin, Germany, 25–26 March 2000*, volume 33. Elsevier, Amsterdam, 2000.

[49] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines - A lightweight formal approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000), Proceedings*, volume 1783 of *LNCS*. Springer, 2000.

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

[51] M. Rößiger. From modal logic to terminal coalgebras. *Theoretical Computer Science*, 260:209–228, 2001.

[52] J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. *Journal of Universal Computer Science*, 7(2):175–193, 2001.

[53] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, 1999.

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

[55] G. Schneider and J. P. Winters. *Applying Use Cases : Second Edition*. Addison-Wesley, 2001.

[56] K. Siau and T. Halpin, editors. *Unified Modeling Language: Systems Analysis, Design and Development Issues*. Idea Group Publishing, 2001.

[57] J. E. Smith. *UML Formalization and Transformation*. PhD thesis, Northeastern University, Boston, Massachusetts, 1999.

[58] D. A. C. Soon-Kyeong Kim. Formalizing the UML Class Diagram Using Object-Z. In R. France and B. Rumpe, editors, *UML'99- Second International Conference on the Unified Modeling Language: Beyond the Standard*, volume 1723 of *LNCS*, pages 83–98. Springer, 1999.

[59] M. Sun and B. Aichernig. Component-based Coalgebraic Specification and Verification in RSL. Technical Report 267, UNU/IIST, October 2002.

[60] M. Sun and B. Aichernig. $\mathbf{Coalg}_{KPF}$: Towards a Coalgebraic Calculus for Component-based Systems. Technical Report 271, UNU/IIST, January 2003.

[61] H. Tews. Coalgebras for binary methods: Properties of bisimulations and invariants. *Theoretical informatics and applications*, 35(1):83–111, Feb. 2001.

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

[63] The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall International, 1992.