Formal Redesign of UML Class Diagrams

Piotr Kosiuczenko

Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67
D-80538 München, Germany
kosiucze@informatik.uni-muenchen.de

Abstract: In this paper we study a formal concept of redesign for object-oriented specifications. This concept corresponds to the UML notion of abstraction. The notion of refinement, which has been extensively studied also at the formal level, models well incremental approach where new requirements are added, but can not be changed. This assumption is usually not satisfied in software engineering process where permanent change is a constant factor. We study therefore a new notion which generalizes the notion of interpretation used in algebra. This notion is very flexible and allows us for comparison of different class diagrams even if one of them contains requirements excluded by another. To compare specifications, we map model elements in the first specification on the related model elements in the second specification. This mapping defines a UML trace; it can be lifted to the level of OCL as well as to the level of first order logic and then extended to an interpretation function. We also provide a formal foundation for our concepts and prove its soundness. We demonstrate the applicability of our approach in a series of examples.

1 Introduction

OO modelling languages provide textual and diagrammatic means for system specification (cf [UML99]). An object oriented system and its real-world environment are modelled using a set of abstractions like class, association, generalization, operation or property. Class diagrams describe the structure of the system specifying common structure and behavior of a set of objects as well as their relationship. The model of the real-world is then extended to the system model specifying the system structure. This happens in a series of steps.

In 80‘ties and in the first half of 90‘ties, the software engineering processes was in a sense incremental and it was hard to modify requirements. For example, in the waterfall model one has to start with a correct requirement specification which then has to be refined to design specification, and the design specification has to be implemented. These steps can be well described using the notion of refinement. In the area of object-oriented software engineering there have been several formal notions of refinement studied (cf e.g. [La95, Le97, PR94], see also [EK99]).

This works fine, if the requirements do not change and the software developers have a clear idea how to proceed. But in practice, a specification is not only extended, but constantly changes due to several factors like changed or new requirements, new technology enablers and so on. In such a case an extensive reengineering of system specification, design or implementation is needed; this implies redesign and reimplementation of the class structure.

While changing the specification or design one has to be aware which properties have to be preserved and what is irrelevant. The notion of refinement with its monotonicity assumption can hardly model such changes. Unfortunately, the research on formal foundations of class structure redesign is underdeveloped.

In this paper we are going to preset a formal notion of interpretation for UML class diagrams with OCL constraints [UML 99, WK99]. Our concept differs from the idea of using transformation rules describing what can be done (such rules may have the form of graph rewriting rules for example, cf e.g. [Gr99]). Our concept is motivated by the notion of abstraction as it is used in UML which allows to relate model elements in different specifications. It generalizes the notion of interpretation used in abstract algebra [Ta73]; the idea is to relate structures with the same behavior but possibly different signatures. Interpretation and refinement functions usually concern all properties, but in a redesign some properties may be intentionally neglected, moreover one specification may contain requirements which contradict requirements in the other. This is well modelled by the partiality of the interpretation function. To compare different specifications selected model elements in the first specification are mapped on the related model elements in the second one. Such a mapping can be first lifted to a formalization of the specifications in the first order logic and extended to an interpretation function. We provide a sufficient condition guaranteeing existence of such an interpretation function. Moreover, this function can be directly defined on the OCL level. In general, the idea is that an implementer has to specify explicitly the trace, i.e. those properties which have to be preserved, and if necessary, the implementer has to provide a proof that these properties are really satisfied, since it is not automatically guaranteed by existence of the interpretation function.

Our notion applies also to state machines which are the basic mean for describing an object behavior in UML. There are several possibilities to implement state machines, for example states can be implemented by enumeration types or by classes. We show how to compare such implementations in an abstract way.

The formalization of OCL we use here comes from [BHTW99] and is performed in CASL [CoFI98]. But in general we do not restrict our concepts to any particular OCL formalization. Let us point out, that the notions developed here can be used beyond the realm of formal methods. Nevertheless, the formal framework is necessary to provide an unambiguous foundation for our approach and prove its soundness. An important feature of this formal approach is that it can be combined with different methods and techniques (like for example Refactoring, cf [Fo00]). Those methods are usually informal and often lack appropriate tool support. Let us point out that our approach is not biased at any particular method.

We illustrate our approach with a series of examples: In the first one we transform a navigation path. In the second example we redesign an inflexible class structure using the role pattern (cf e.g. [Bä00]). In the third example, we show how to eliminate multiple inheritance from a class diagram. In the fourth example, we describe a very simple flip-flop game using state machines. The states are implemented by enumeration types and then by state classes instead of enumeration types.

2 OCL syntax and semantics

2.1 OCL syntax

In this subsection we define a context free grammar for restricted part of OCL.

```
Identifier   :=  Var | ClassName | self

BTerm        :=  b | BTerm[and | or | implies] BTerm  | not BTerm |
                 Term [= | < | >] Term |
                 oclIsTypeOf(ClassName)  | oclIsKindOf(ClassName) |
                 Term -> forAll(BTerm) | Term -> exists(BTerm) |
                 Term -> isEmpty


Term         :=  Identifier | Identifier.b | BTerm |
                 if BTerm then Term else Term |
                 n | Term [+ | - | * | /] Term |
                 Term.qop (Term,..., Term)
```

The nonterminal *BTerm* defines the boolean valued OCL operations, in particular *b* stands for boolean values like `true` or `false`. The nonterminal *Term* defines the set of other OCL terms and their composition as well as constants like integers or reals (denoted by the nonterminal *n*). The OCL terms can be translated to terms in the sense of algebra, this allows for a formal semantics (see below).

2.2 OCL semantics

In this subsection we present shortly an OCL semantics which is a slight modification of [BHTW99], the underlying algebraic notions can be found in the appendix. The basic OCL types `Boolean`, `String`, `Integer`, `Real` are modelled in our semantic by the sorts Boolean, String, Integer, Real. Every OCL operation defined on this types is modelled by a corresponding function. An element is of sort collection iff it is of sort Set or of sort Bag or of sort Sequence. We assume that singleton sets, bags or sequences, i.e. sets, bags or sequences containing one element only, are identified with their elements. This assumption is often made in algebraic specification and does not lead to a contradiction.

Every function f : A —>B is lifted to the corresponding sets f : Set(A) —>Set(B) in a natural way f(X) = {f(x) | x ∈ X)} similarly for bugs and sequences. We use boolean valued functions _->exists(o | ...), _->forAll(o | ...), _->isEmpty,... for the equally named OCL operations on collections types (the definition is straightforward).

The environment is modelled by the sort Env. We add also a sort for class names ClN and for each class C ∈ ClN an equally named sort C of object identifiers. The set of all object id's belonging to all classes is denoted by Id. The current values of object attributes are defined by a pair <e, o>, i.e. we have a pair constructor <_, _> : Env × C —>Env_C for each class C, as well as the two corresponding selectors:
_.Env : Env_C —>Env and _.C : Env_C —>C specified by the equations:

<e, o>.Env = e, <e, o>.C = o, <x.Env, x.C> = x.

Moreover, we require that the constructor distributes with the set theoretical union:

<e, x ->union(y) > = <e, x> ->union(<e, y>).

For every class C, there is a function _oclIsTypeOf(_) : Env_Id × ClN —> Boolean such that <e, o>.oclIsTypeOf(C) evaluates to true iff the object `o` in the current environment `e` is of class `C`, but not of any of its subclasses. This function corresponds to `o.oclIsTypeOf(C)`. The OCL expression `o.oclIsKindOf(C)` states that `o` is of class `C` or any of its subclasses; this operation is modelled by the term
oclIsKindOf() : Env_Id × ClN —>Boolean .

Every operation `op(`x_1`:` T_1`,…,` x_n`:` T_n`)` returns new environment and optionally a value. An operation or property returning a value of type T is modelled by a function of the form op : Env_C × T_1 ×…× T_n —>Env_T. We need the Env component since `op` may modify the environment. If the operation is a query, then the environment component is kept unchanged. Attributes and associations are treated as query operations. If `a` is an object attribute, then the function _.a returns the corresponding value (i.e. <e, o>.a). Similarly, we model associations between classes as functions, if `lnkB` is a directed association from class A to class B, then we model `lnkB` as a function _.lnkB : A_Env —>Set(B).

To interpret OCL invariants we define interpretation function Trans by structural induction:

Trans(`self`) $=_{df}$ self

Trans(`u.a`) $=_{df}$ <env, Trans(u)>.a, for an OCL term `u`

Also an OCL term of the form `self.`a_1. … .a_{n-1}.a_n is translated to the term of the term a_n(env, a_{n-1}(env,...a_1(env, self)...)) written in the standard prefix notation. Trans is assumed also to preserve operations on the basic OCL types like equation. The suffix '.T' means that we select the value.

On the other hand, it is not hard to observe that the Trans has an inverse function $Trans^{-1}$ such $Trans^{-1}$(Trans(u)) = u, for all OCL terms u. This function allows us to translate algebraic terms, as introduced in this section, into OCL terms.

Given constraint of the form ([:T] means that T is optional):

```
context C :: op(x1: T1,…, xn: Tn)[:T]
inv : Ψ
```

This constraint is formalized by the formula:

$$\forall \text{ env: Env, self : C, } x1 : T_{1, \ldots, xn} : T_n \, [_{\text{, result :T}} \text{ result} = \text{<env, self>.op}(x_1, \ldots, x_n) \wedge] \\ \text{Trans}(\Psi)$$

2.3 Example

In this example, we show how to translate OCL formulas to the first order logic. An oo-database stores information about students and the staff working in a faculty (see figure 1). The abstract class `Person` is extended by the class `Student`, `Assistant` and `Professor`. The class `Professor` is extended by the class `Dean`.

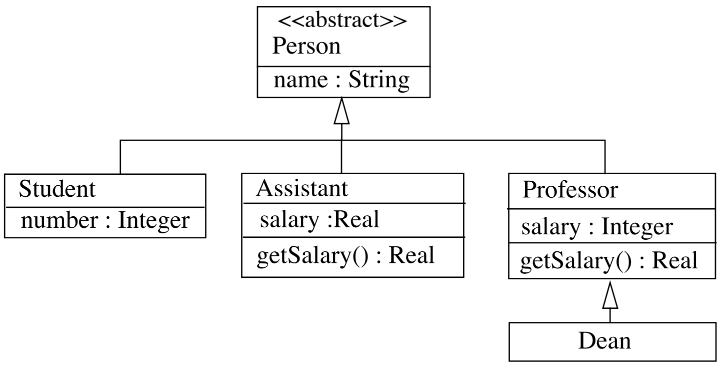

Figure 1: Role classes

We assume that the following OCL formulas have to be satisfied:

```
context Assistant inv:
self.salary > 0 and self.salary = self.getSalary()
```

Similarly for the class `Professor`. We express selected class dependencies of this diagram using OCL formulas. The property that `Person` is an abstract class can be expressed by the following invariant:

```
context Person inv:
not(self.oclIsTypeOf(Person))
```

This diagram implies that each `Dean` is a `Professor`. This property can be expressed in OCL as follows:

```
context Dean inv:
self.oclIsKindOf(Professor)
```

We may formulate this in a different way, namely we may say that each person who is a dean must be a professor:

```
context Person inv:
self.oclIsKindOf(Dean) implies self.oclIsKindOf(Professor)
```

This property is weaker then the previous one, and relativizes the previous statement to the `Person` class. For example, it would be possible that an object, which does not instantiate the class `Person`, was a dean without being a professor. Let us observe that the first constraint implies that the class `Professor` is navigable from the class `Dean`, but the second is not (cf example 5.2).

We formalize the above OCL formulas as follows: We introduce sort names Person, Student, Assistant, Professor, Dean for the classes in the diagram. The attributes like `number` in the class `Student`, `salary` in the classes `Assistant` and `Professor` are formalized by functions:
_.number : Env_Student —>Integer
_.salary : Env_Assistant —>Integer,
_.salary : Env_Professor —>Integer, resp.

The query operations are formalized by functions:

_.getSalary() : Env_Assistant —>Integer
_.getSalary() : Env_Professor —> Integer

The property that `Person` is an abstract class and that every person which is a dean is a professor are formalized by the formula Φ_1 of the form:

$$\forall_{\text{env : Env, self : Person}} \text{not}(\langle\text{env, self}\rangle.\text{oclIsTypeOf(Person)}) \wedge$$
$$(\langle\text{env, self}\rangle.\text{oclIsKindOf(Dean)} \Rightarrow \langle\text{env, self}\rangle.\text{oclIsKindOf(Professor)})$$

If we formalize the salary constraint for the class Assistant, then we get the formula Φ_2 of the form:

$$\forall_{\text{env : Env, self : Assistant}} \langle\text{env, self}\rangle.\text{salary} > 0 \wedge \langle\text{env, self}\rangle.\text{getSalary()} =$$
$$\langle\text{env, self}\rangle.\text{salary}$$

Similarly for the salary constraint in the class `Professor`.

3 Interpretation function

In this section we introduce a new notion of interpretation. We define it on the level of first order logic, but it can be defined on the level of OCL expressions, thanks to the existence of the function Trans^{-1}. To compare different specifications, selected model elements in the first specification are mapped on the related model elements in the second one. Let us point out that our notion has much weaker properties then the notion of interpretation used in abstract algebra [Ta73], since we assume that the underlying function is partial and preserves only selected properties, i.e. we do not require that it preserves the satisfaction relation in general, but we assume that it preserves composition of terms and the basic OCL types. Interpretation and refinement functions usually concern all properties, but during redesign some properties may be intentionally neglected, moreover one specification may contain a requirements which contradicts requirements in the other. These weaker assumptions provide sufficient flexibility in relating different class structures, they allow us for example to skip those parts of a refined structure which prove to be irrelevant. This is well modelled by the partiality of the interpretation function.

Let A and B be sets of terms. We say that A *spans* B iff A is contained in B and every non-variable term of B can be obtained from terms belonging to A by variable renaming and term composition (see the appendix). The set A is a *base* of B iff in addition every term $t \in B$ can be obtained in a unique way by renaming and composing terms from A.

We call a partial function $\varphi : T(\Sigma, X) \longrightarrow T(\Sigma', X)$ *compositional* iff for all terms t the following conditions hold

- $var(\varphi(t)) \subseteq var(t)$
- if t is of the form $t_0[t_1/x_1,..., t_n/x_n]$, φ is defined on terms t_i, for i= 0,..., n, and $t_0',..., t_n'$ are the corresponding values, then φ is defined on t and $\varphi(t) = t_0'[t_1'/x_1,..., t_n'/x_n]$.

The first condition eliminates some pathological cases. The second one is a compositionality requirement which allows to extend a mapping to a partial function. For example, if Σ has the form $(S, F, \leq)$, then the set $\{f(x_1,..., x_n) \mid f : s_1,...,s_n \rightarrow s \in F\}$ is a base of $T(\Sigma, X)$. This function could be also defined using the forget, restrict and identify operations, but this would complicate the definition and proof.

The following statement justifies the name ‘base‘. It can be easily proved by structural induction.

Statement
Let B be set of terms and let A be a base of B. Every term valued mapping on A can be uniquely extended to a function on B; moreover if B is closed on composition of terms, then the extended mapping is compositional.

For example, every interpretation function in the sense of abstract algebra [Ta73] can be defined as a total interpretation function which uniquely extends a mapping on the set $\{f(x_1,..., x_n) \mid f : s_1,...,s_n \rightarrow s \in F\}$.

In the following, all the redesign examples rely on the theorem below that provides a sufficient condition for a set to be a base. It says that a set of terms generates a base, if it does not simultaneously contain a term and its subterm. The notion of decomposition does not depend on variable names, therefore in the theorem we abstract from variables using a variable renaming and in the proof we prune of the leafs corresponding to variables.

Extendability theorem

Let A be a set of terms such that for all pairwise different terms $t_1, t_2 \in A$ and all variable renamings σ_1, σ_2, $t_1^{\sigma 1}$ is not a subterm of $t_2^{\sigma 2}$. Then A is the base of the set B obtained by composing terms of A.

Sketch of the proof

We can represent each term $t \in B$ as a tree where nodes correspond to function symbols and where variables are not represented. A decomposition of t corresponds to a coloring of the corresponding tree, where colors correspond to the elements of A.

We prove this theorem by contradiction. Let us suppose that A is not the base of B, then there must exist a non variable term $t \in B$ with two different decompositions. We can assume that t has minimal height, i.e. there exist no other term with two different colorings having height smaller than the height of t. Since there exist two different decompositions of t there must exist two different colorings of t. If all the leafs have the same colors, then in both trees we can prune of the corresponding subtrees and obtain a tree of a smaller height than t. Therefore there exist subtrees t´, t´´, such that t´, t´´ share the same leaf but have different colors. Since t´, t´´ are subtrees of the same tree, t´ is a subtree of t´´, or vice versa. This yields the contradiction with the assumption that in A there are no two different terms such that one of them is a subterm of the other. ♦

To compare different class structures compositionality is not enough and the interpretation function has to preserve more properties. We call a partial function $\varphi : T(\Sigma, X) \longrightarrow T(\Sigma', X)$ *interpretation function* iff the following conditions are satisfied:

- φ is compositional
- for every term t of a basic OCL type s, $\varphi(t)$ is of type s too, if defined
- if t has the form $f(x_1,..., x_n)$ where f is a predefined OCL operation, such that the variables are of the basic OCL types, then $\varphi(t) = t$

The second condition requires that φ preserves basic OCL types. The last one requires that φ preserves predefined operations on the basic OCL types. Interpretation function would not make much sense, if not extended to formulas. This can be done by structural induction:

Let $Sp = Spec(\Sigma, Ax)$, $Sp' = Spec(\Sigma', Ax')$ be specifications and let $\varphi : T(\Sigma, X) \longrightarrow T(\Sigma', X)$ be an interpretation function. If $\varphi(\Phi) = \Phi'$ and $\varphi(\Psi) = \Psi'$, then $\varphi(\Phi \wedge \Psi) =_{def} \Phi' \wedge \Psi'$, similarly for other boolean operations. If $\varphi(\Phi) = \Phi'$, then $\varphi(\forall_x \Phi)) =_{def} \forall_x \Phi'$, similarly for the existential quantifier.

Let us remind that we do not assume that interpretation function preserves satisfaction relation in general, as in the case of institutions, but only selected properties. As in the case of institutions we do not want to restrict the notion of satisfaction. We say that φ *preserves* specification Sp, if φ is defined on Ax and $\varphi(\Phi)$ is valid in Sp´, for every $\Phi \in Ax$.

As we mentioned, an interpretation function can be lifted to OCL terms. Given an interpretation function φ such that φ maps class `C` on another class `φ(C)` and an OCL constraint of the form

```
context C inv:
Ψ
```

let us define $\phi(x) =_{df} \mathrm{Trans}^{-1}(\varphi(\mathrm{Trans}(x))$. We obtain a new OCL constraint:

```
context ϕ(C) inv:
ϕ(Ψ)
```

4 Redesign

The concept of redesign is more general then the concept of refinement. We do not assume that properties are added or refined, but we allow to change them in an arbitrary way as long as some selected properties are kept unchanged, for example a number of design level classes might be restructured or a specification level class might be split into several design level classes. Preserved properties may concern dependencies between classes, associations, operations, or generalization relation.

In UML [UML99], dependency is a term for a directed relationship from a client to a supplier stating that the client depends on the supplier. The notion of redesign studied here corresponds to the notion of abstraction in UML. Abstraction is a kind of dependency which relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints. There are four standard stereotypes for the abstraction dependency: «derive», «realize», «refine» and «trace». The derived abstraction specifies that the client may be computed from the supplier. The realize abstraction is a mapping between specification model elements (the supplier) and model elements that implement it (the client); the implementation model elements are required to support the operations that the specification model element declares. The refine abstraction specifies refinement relationship between model elements at different levels, such as analysis and design. The trace abstraction specifies a trace relationship between model elements that represent the same concept in different models. Traces are used for tracking requirements and changes across models.

Usually while redesigning or implementing a specification one has an intuitive idea what a the trace of one specification in the other is. For example, if in the initial and in the latter specification two equally named classes exist, then the class in the second specification is meant to implement or redesign the class in the first one; similarly when in such classes equally named operations exist. Below we show that one can map model elements on model elements and that such a mapping can be formalized as an interpretation function. In the UML metamodel a mapping is an expression that is used for mapping model-elements in one diagram on model elements in another diagram. A mapping expression states an abstraction relationship between the supplier and the client. It may be formal or informal and unidirectional or bidirectional. The idea of this approach is that a mapping has to be defined on its ‘generators‘, and that such an implicitly defined mapping can be then extended to an interpretation function on the level of OCL specifications and on the level of formal specifications, if the assumption of the extendability theorem is satisfied. The applicability of this theorem can be easily checked even at the informal OCL level. The designer or implementer have to provide explanation how these selected properties are reflected in the new class structure by providing a mapping.

$$\begin{tikzcd}
\text{ModelElements} \arrow[r, "\text{mapping}"] \arrow[d] & \text{ModelElements}' \arrow[d] \\
\text{OclSpec} \arrow[r, dashed, "\text{interpretation}"] \arrow[d, shift right, "\text{Trans}"'] & \text{OclSpec}' \arrow[d, shift right, "\text{Trans}"'] \\
\text{FormalSpec} \arrow[r, dashed, "\text{interpretation function}"] \arrow[u, shift right, "\text{Trans}^{-1}"'] & \text{FormalSpec}' \arrow[u, shift right, "\text{Trans}^{-1}"']
\end{tikzcd}$$

Figure 2: Logical relation

The figure above shows the relation between introduced concepts. Model elements are implicitly modelled by OCL terms. OCL constraints are translated to first order logic, as we have shown in section 2. A mapping between model elements generates an interpretation function on the level of OCL and on the level of formal specifications, if the condition of the extendability theorem is satisfied. In such a case the diagram above commutes.

The interpretation function is very flexible, we will show that this notion allows one to treat not incremental changes of class diagrams. Since it does not preserve the subtype relation, the generalization relationship is in general not preserved, but it can be modelled by other dependencies. We can for example model or implement inheritance by associations (cf e.g. [GR99]). The interpretation function may map a query or an association on a composition of associations, if they are navigable. Similarly, an operation can be delegated to another class, if there is a path from one of the classes to the other.

In general, if a property has to be preserved, then all navigability properties implicitly implied by this property must be preserved too. We will say that a class is navigable from another class if there exists a traversable sequence of classes connected by associations, query operations which return objects from another class and generalization relationship. The navigability is implicitly preserved by the interpretation function, provide that the function is defined on all parts of the navigation path.

Let us point out here that of course we are not the first ones who consider restructuring of UML specifications with OCL constraints (cf e.g. [DW98]), but to our best knowledge we are the first who do it at the formal level.

5 Applications

5.1 Example: Navigation path

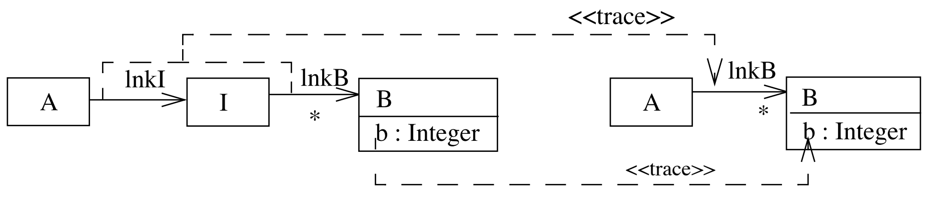

Figure 3: Navigation path redesign

In this subsection we consider a redesign of a navigation path. The path `lnkI.lnkB` on the left hand side of figure 3 is reduced to `lnkB` by deleting the intermediate class `I`. Such a redesign can be easily treated in our framework.

We map class `A` of the first specification on class `A` of the second one, similarly we map class `B` on class `B`, `lnkI.lnkB` on `lnkB`, and the attribute `b` in the first specification on the attribute `b` in the second specification. On the formal level class names A, B are mapped on A, B, respectively. The term <env, <env, self : A>.lnkI>.lnkB : B is mapped on the term <env, self : A>.lnkB : B, and the term <env, self : B>.b : Integer in the first specification on the term <env, self : B>.b : Integer in the second one.

This mapping induces an interpretation function, since the condition of the extendability theorem is satisfied. Let us observe, that also the inverse mapping can be extended to an interpretation function.

5.2 Example: Role pattern

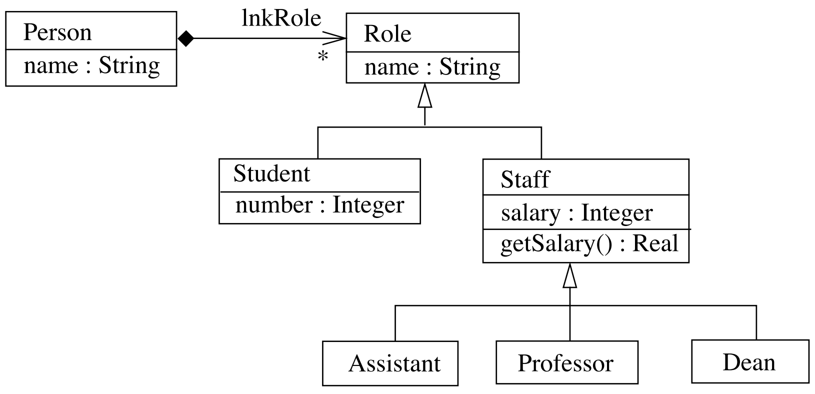

Figure 4: Role pattern

Let us consider the class structure described in example 2.3 (cf figure 1). It is well known that this structure makes hard any change of person‘s class, it also disallows a single person to play multiple roles. Therefore, we use the much more flexible role pattern (see figure 4, cf e.g. [Bä00]). It would not be reasonable to assume that the class `Person` is abstract and only its subclasses can be instantiated, but we may mirror this property by requiring that each person must play a role. In the previous specification it was automatically guaranteed that a dean is a professor. In this class diagram it is not guaranteed any more, therefore we add a constraint that each person being a dean is a professor:

```
context Person inv:
not(self.lnkRole->isEmpty)   and
self.lnkRole->exists(r | r.oclIsKindOf(Dean)) implies
      self.lnkRole->exists(r | r.oclIsKindOf(Professor))
```

The following constraint is analogous to the salary-constraints in example 2.3.

```
context Staff inv:
self.salary > 0 and self.getSalary() = salary
```

Let Φ_1' be the formalization of the first OCL constraint, then Φ_1' has the form:

$$\forall_{env : Env,\ self : Person}\ \neg(\text{<env, self>.lnkRole->isEmpty}) \wedge \text{<env, self>.lnkRole->exists(r | <env, r>.oclIsKindOf(Dean))} \Rightarrow \text{<env, self>.lnkRole->exists(r | <env, r>.oclIsKindOf(Professor))}$$

The formalization of the second OCL constraint yields a formula Φ_2' of the form:

$$\forall_{env : Env,\ self : Staff}\ \text{<env, self>.salary} > 0 \wedge \text{<env, self>.getSalary()} = \text{<env, self>.salary}$$

Let us analyze the relation between these two class structures. We map the term (x : Env_Person).oclIsKindOf(Student) on the term
(x : Env_Person).lnkRole -> exists(o|<env, o>.oclIsKindOf(Student)),
similarly for the classes `Assistant`, `Professor` and `Dean`. The terms corresponding to attributes and queries are mapped on the corresponding terms in the formalization of the second specification, for example the term _.salary : Env_Assistant —> Integer is mapped on the term _.salary : Env_Assistant —> Integer in the second specification. We interpret x.oclIsTypeOf(Person) as x.lnkRole->isEmpty. It is not hard to prove that the condition of the extendability theorem is satisfied, therefore the mapping can be extended to an interpretation function φ.

Let us observe, that the interpretation of the formula Φ_1 from example 2.3 coincides with Φ_1', i.e. $\varphi(\Phi_1) = \Phi_1'$. The interpretation of Φ_2 is not equal to Φ_2', but has the form :

$$\forall_{env : Env,\ self : Assistant}\ \text{<env, self>.salary} > 0 \wedge \text{<env, self>.getSalary()} = \text{<env, self>.salary}$$

The formula Φ_2' implies $\varphi(\Phi_2)$, since the second one is relativized to a smaller set Assistant and since in an order sorted algebra, a function on supersorts and subsorts must agree (see the appendix). Consequently, the interpretation function φ preserves the initial specification.

5.3 Example: Multiple inheritance

There are many ways to resolve multiple inheritance; one possibility is to use delegation (cf e.g. [GHJW95, GR99]). Let us consider the following example: Class `C` inherits from classes `A` and `B` (see the left hand side of figure 5). We require also that the following OCL constraint holds:

```
context B inv:
self.getB() = b
```

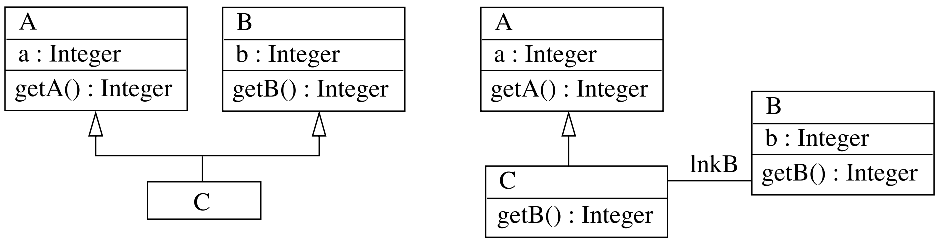

Figure 5: Resolving multiple inheritance

One can not directly implement this class structure in Java, therefore a redesign is needed. In the redesigned structure, class `C` inherits from classes `A` and delegates operation `getB()` to class `B`.
The second diagram is augmented with the following constraints:

```
context B inv:
self.getB() = self.b
```

```
context C inv:
self.getB() = self.lnkB.getB()
```

The classes `A`, `B` and `C` with their attributes and operations are mapped on the corresponding model elements in the second specification respectively. The inherited operations `getA()` and `getB()` of class `C` in the first specification are mapped on the inherited operation `getA()` and the new operation `getB()` of class `C` in the second specification. Formally:

x.oclIsKindOf(X) is mapped on x.oclIsKindOf(X), for X = A, B, C.

<env : Env, self : A>.a : Integer is mapped on <env : Env, self : A>.a Integer

<env : Env, self : A>.getA : Integer is mapped on <env : Env, self : A>.getA : Integer

...

<env, self : C>.b : Integer is mapped on <env, <env, self : C>.lnkB>.b : Integer

<env, self : C>.getB : Integer is mapped on <env, self : C>.getB : Integer

Let us observe, that the interpretation function generated by this mapping does not preserve the subsort relation and in particular that two function are treated as different if they types differ, what makes interpretation function work (cf the appendix). It is also not hard to observe that the generated function preserves the first OCL constraint.

This mapping can be reversed. Let us extend the inverse mapping by mapping:
<env, <env, self>.lnkB>.getB() : Integer on <env, self>.getB() : Integer.
The extended mapping satisfies the extendability condition, since `lnkB` was not mapped on any element in the first specification. The second OCL invariant in the second specification is mapped on the trivial invariant:

```
context C inv:
self.getB() = self.getB()
```

5.4 State machines, pre- and post-conditions

State machines are the basic mean to describe object behavior in UML. In object-oriented modelling there exist several approaches to implement state machines. For example states in a state machine can be implemented by enumeration types or by state classes as in the State Pattern [GHJW95]. The problem arises how to compare such seemingly different implementations. On the other hand, there exists several formal semantics of state machines (cf e.g. [RACH00]) which at the first glance seem to concern a very specific case or class structure. This approach allows to compare them and treat these semantics in a more abstract way.

In this section we apply our concept to a simple state machine example. First we implement states by using enumeration type, then using object classes. The state machine is constraint by pre- and post- conditions. Pre- and post-conditions can be formalized in similar way as invariants, but one has to take care of different system states before and after execution of the operation. Given a constraint of the form:

```
context C :: op(x1: T1,…, xn: Tn)[:T]
pre  : Ψ
post : Ψ´
```

To translate the pre-condition use the function Trans, and to translate the post-condition we use the function $\text{Trans}_{\text{post}}$ [BHTW99] which is defined as Trans, but Transpost(t.a) = <env´, $\text{Trans}_{\text{post}}$(t)>.a and $\text{Trans}_{\text{post}}$(t.a@pre) = <env, $\text{Trans}_{\text{post}}$(t)>.a, where `a` is a property returning value of type T and env´ is a new variable. The variables env, env´ used by functions Trans and $\text{Trans}_{\text{post}}$, model the environments before and after the execution of the operation `op`, respectively. Above constraint is formalized as follows:

$$\forall_{\text{self : C, env, env´ : Env, x1 : }T_{1,\ldots,\text{ xn}}\text{ : }T_n\text{ [, result : T}} \text{ result} = \langle\text{env, self}\rangle.\text{op}(x_1,\ldots, x_n).\text{T} \wedge \text{]}$$
$$\text{env´} = \langle\text{env, self}\rangle.\text{op}(x_1,\ldots, x_n).\text{Env} \wedge \text{Trans}(\Psi) \Rightarrow \text{Trans}_{\text{post}}(\Psi´)$$

In this example we consider a state machine diagram for the class FlipFlop. An object of this class can be in the state `flip` or `flop`. There exists an operation `next()` which changes one states to the other (see figure 6).

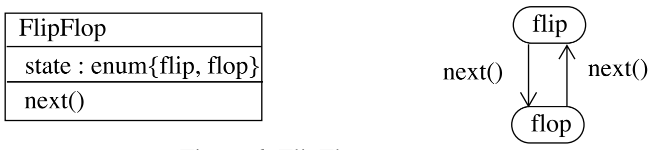

Figure 6: FlipFlop game

The behavior of this state machine can be specified in OCL as follows:

```
context FlipFlop :: next()
post: state@pre = #flip implies state = #flop and
        state@pre = #flop implies state = #flip
```

This constraint is formalized as follows

$$\forall_{\text{self : FlipFlop, env, env}' \text{ : Env}} \text{env}' = \text{<env, self>.next().Env} \Rightarrow$$
$$(\text{<env, self>.state = flip} \Rightarrow \text{<env}', \text{self>.state = flop} \wedge$$
$$\text{<env, self>.state = flop} \Rightarrow \text{<env}', \text{self>.state = flip})$$

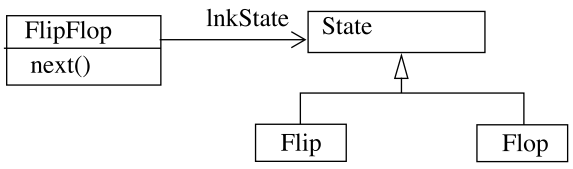

Figure 7: States as classes

We may redesign this state machine using the State Pattern. In the redesigned version, the states are modelled by objects of the class `State`, which has two subclasses `Flip` and `Flop` (see figure 7).

This class diagram is augmented with the following OCL constraint:

```
context FlipFlop :: next() post:
lnkState@pre.oclType = Flip implies lnkState.oclType = Flop
     and
lnkState@pre.oclType = Flop implies lnkState.oclType = Flip
```

This constraint is formalized as follows:

$$\forall_{\text{self : FlipFlop, env, env}' \text{ : Env}} \text{env}' = \text{<env, self>.next().Env} \Rightarrow$$
$$(\text{<env, self>.lnkState.oclType = Flip} \Rightarrow \text{<env}', \text{self>.lnkState.oclType = Flop} \wedge$$
$$\text{<env, self>.lnkState.oclType = Flop} \Rightarrow \text{<env}', \text{self>.lnkState.oclType = Flip})$$

We map the elements of the enumeration class on the corresponding classes i.e. `#flip`, `#flop` are mapped on classes `Flip` and `Flop`, respectively, and the attribute `state` is mapped on the operation `oclType`. It is not hard to observe that this mapping, defined on the level of model elements, induces an interpretation function.

6 Concluding remarks

In this paper, we have developed a simple and practically useful approach to formal redesign of various kinds of class structures with OCL constraints. It allows us to relate dissimilar class structures and to define the trace relationship between model elements representing the same concepts in such structures. We have provided a formal foundation for this approach and proved its soundness.

Nevertheless their is still a lot to be done.We have to investigate means allowing one to avoid tedious definitions of mappings. As pointed out by the anonymous referees, defining a mapping, even on the generators only, is too tedious for real size systems, therefore one needs additional means like mapping equally names elements on equal named elements. We have also to extend our approach to allow us for comparison of different UML models corresponding to different views. We have to study the properties of the interpretation function in logical terms in particular the relation to institutions (cf [Ta99]). And last but not least, we plan to implement a tool supporting class redesign. Such a tool would be very helpful since a purely manual transformation of complex OCL constraints is very laborious and error prone.

Acknowledgment

It is a pleasure to thank Martin Wirsing for his advice during the preparation of this paper, as well as Hubert Baumeister and the anonymous referees for their helpful comments.

References

[Bä00] Bämer, D. et. al.: Role Object. In (Harrison, N.; Rohnert, H. Eds): Pattern Languages of Program Design. Addison-Wesley, 2000.

[BHTW99] Bidoit, M.; Hennicker, R.; Tort, F.; Wirsing, M.: Correct realizations of interface constraints with OCL. The UML - Beyond the Standard, Proc. 2nd Int. Conf., UML'99, LNCS 1723, Springer, Berlin, 1999, pp 399-415.

[CoFI98] CoFI Task Group Language Design. CASL - The CoFI algebraic specification language. http://www.brics.dk/Projects/CoFI/.

[DW98] D. D´Suza, A. Wills. Objects, Components, and Frameworks with UML, The Catalysis Approach. Addison Wesley, 1998.

[EK99] Ehrig, H.; Kreowski, H. J.: Refinement and implementation. In (Astesiano, E.; Kreowski, H. -J.; Krieg-Brückner B. Eds): Algebraic Foundations of System Specification, Springer 1999.

[Fo00] Fowler, M.: Refactoring: improving the design of existing code. Reading, Mass., Addison-Wesley, 2000.

[GHJW95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Addison-Wesley, Reading, 1995.

[GR99] Gogolla, M.; Richters, M.: Martin Gogolla and Mark Richters. Transformation Rules for UML Class Diagrams. In (Bezivin J., Muller P. A., Eds.):, Proc. 1st Int. Workshop Unified Modeling Language (UML'98), pp 92-106.

[GM92] Goguen, J., Meseguer, J.: Order sorted algebra. Theoretical Computer Science, vol. 105(2), Elsevier, Amsterdam, 1992, pp 167-215.

[Gr99] Große-Rhode, M.et. al.: Refinements and modules for typed graph transformation systems. In (Fiadeiro, J. Ed.): Algebraic Development Techniques, ETAPS´98, LNCS 1589, Springer, Berlin, 1999, pp 137-151.
[La95] Lano, K.: Formal object oriented development. Springer, Berlin, 1995.
[Le97] Lechner, U.: Object-Oriented Specification of Distributed Systems. Disser-tation, Universität Passau, Fakultät für Mathematik und Informatik, 1997.
[PR94] Paech, B.; Rumpe, B.: A new concept of refinement used for behavior modeling with automata. In Proceedings of FME´94, LNCS 873, Springer, Berlin, 1994.
[RACH00] Reggio, G.; Astesiano, E.; Choppy, C.; Hussmann, H.: Analyzing UML Active Classes and Associated State Machines - A Lightweight Formal Approach. In (T. Maibaum Ed.): Proc. FASE 2000, LNCS 1783, Springer, Berlin, 2000.
[Ta99] Tarlecki, A.: Institution: An Abstract Framework for Formal Specifications. In (Astesiano, E.; Kreowski, H. -J.; Krieg-Brückne, B. Eds.): Algebraic Foundations of System Specification, Springer 1999.
[Ta73] Taylor, W.: Characterizing Malcev conditions. Algebra Universalis, vol. 3, Springer, Berlin, 1973, pp 351-397.
[UML99] OMG. Unified Modeling Language Specification. Version 1.3, June 1999.
[WK99] Warmer, J.; Kleppe, A.: The Object Constraint Language. Addison-Wesley, Reading, 1999.

7 Appendix: signatures, structures and formulas

A many sorted *signature* Σ is a pair (S, F) where S is a set of sorts and F is a set of function symbols paired with their types, i.e. $f : s_1,...,s_n \to s \in F$. An *order sorted signature* Σ is a triple $(S, F, \leq)$, where (S, F) is an algebraic signature and $\leq$ is a partial order on S. The set of all terms with variables in S sorted set $X = (X_s)_{s\in S}$ is denoted by $T(\Sigma, X)$. For a term t, var(t) is the set of variables contained in t. The notation $t(x_1,..., x_n)$ means that t contains at most the variables $x_1,..., x_n$, i.e. $var(t) \subseteq \{x_1,..., x_n\}$. $t_0[t_1/x_1,..., t_n/x_n]$ denotes the term obtained from $t(x_1,..., x_n)$ by simultaneous substitution of t_i for x_i, for i = 1,..., n (this operation is called *term composition*). The terms t_i are called *subterms* of $t_0[t_1/x_1,..., t_n/x_n]$. Term sorts are defined by structural induction: If $t_i : s_i$ and $f : s_1,...,s_n \to s$, then $f(t_1,..., t_n) : s$. Let $A= (A_s)_{s\in S}$, $B = (B_s)_{s\in S}$ be two S-sorted sets, a function $f : A \longrightarrow B$ is an S-indexed family of functions $A_s \longrightarrow B_s$, for $s \in S$. Let $\rho: X \longrightarrow T(\Sigma', X)$ be a mapping, we can extend ρ to $\rho: T(\Sigma,X) \longrightarrow T(\Sigma',X)$ defined by $\rho_s(t(x_1,..., x_n)) =_{def} t(\rho_{s_1}(x_1),..., \rho_{s_n}(x_n))$ where x_i is of sort s_i for i = 1,..., n. ρ is called *variable renaming*, if it maps variables on variables. An *order-sorted algebra* [GM92] $A = ((A_s)_{s\in S}, (f^A)_{f\in F})$ over a signature Σ consists of a family of non empty carrier sets $(A_s)_{s\in S}$ such that $A_s \subseteq A_u$, for $s \leq u$, and a family of functions $(f^A)_{f:s1,...,sn\to s\in F}$ such that $f^A: A_{s_1}\times...\times A_{s_n} \longrightarrow A_s$, for $f : s_1,...,s_n \to s$, moreover if $f : u_1,...,u_n \to u$, where s_i are subsorts of u_i, then both functions must coincide on the sets A_{s_i}. A *specification* is a pair Spec(Σ, Ax) consisting of a signature Σ and a set formulas Ax over the signature Σ. The set Ax can be a set of formulas.