Open access • Journal Article • DOI:10.1007/S10270-007-0068-6

Redesign of UML class diagrams: a formal approach — Source link

Piotr Kosiuczenko

Institutions: University of Leicester

Published on: 01 Apr 2009 - Software and Systems Modeling (Springer-Verlag)

Topics: Applications of UML, Class diagram, UML tool, Code refactoring and Interpretation (logic)

Redesign of UML Class Diagrams: A Formal Approach[1]

Piotr Kosiuczenko

Department of Computer Science
University of Leicester, LE1 7RH, UK

Abstract: Contracts provide a precise way of specifying object-oriented systems. When a class structure is modified, the corresponding contracts must be modified accordingly. This paper presents a method of transforming contracts, which allows the extension of a mapping defined on a few model elements, to - what we call - an interpretation function, and to use this function to automatically translate OCL-constraints. Interestingly, such functions preserve reasoning using prepositional calculi, resolution, equations, and induction. Interpretation functions can be used to trace model elements throughout multiple redesigns of UML class diagrams in both the forward, and the backward direction. The applicability of our approach is demonstrated in several examples, including some of Fowler's refactoring patterns.
Keywords: UML, OCL, formal methods, refactoring, requirements tracing.

1 Introduction

Object-oriented modelling languages provide textual, and diagrammatic means for system specification (cf. [UML05]). An object-oriented system and its real-world environment are modelled using abstractions such as class, association, generalization, operation, and property. Class diagrams specify a common structure, and relationships between objects; they are often detailed by constraints. System specification, design, and implementation occur in a series of steps. There exist different software engineering approaches which detail how those steps can be performed. In the 80's, and early 90's, the waterfall model prevailed in software engineering. In this process, one has to begin with fixed requirement specification, design the system, and then implement the design specification. These steps can be adequately described using the notion of refinement. In the area of formal methods, and object-oriented software engineering several notions of refinement have been studied (cf. e.g. [La95, PR94], see also [EK99]).

The waterfall model works correctly provided the requirements do not change, and the software developers have a clear idea regarding how to proceed. In practice however, a specification is not only extended, but constantly changes due to a number of factors such as changed or new client requirements, new technology enablers, and so on. In such a case extensive re-engineering of system specification and design is needed. The notion of refinement, with its monotonicity assumption, can barely model such changes.

1. This is the corrected version of the technical report.

Requirements management belongs to the crucial activities in the engineering of complex software systems. Existing notions of refinement barely cope with the non-monotonic change of requirements. Contemporary software engineering processes, such as Unified Process, embrace change of the specification, design, and implementation as being a constant factor. In this case, requirements tracing is much harder to achieve. For example, if an interface or signature changes, a formula or a constraint which described a property concerning classes implementing this interface may no longer make sense.

A number of approaches to redesign of UML class models exist already. The best known is the so-called refactoring [Fo00]. This approach provides simple patterns for code, and class structure redesign to extend, to improve, and to modify a system. No tool in the market allows for an automatic transformation of constraints. Today, manual transformation must be performed in order to modify constraints, but this is very time consuming, error prone, and may result in different types of errors ranging from omission of constraints to incorrect, or even inconsistent, transformation. Moreover, one has to manually redo the accompanying proofs to ensure that, for example, an invariant implies a precondition.

In this paper we study the redesign of UML class diagrams with OCL-constraints [UML05, WK99], as well as the transformation, and tracing of constraints. We present a new notion of interpretation function for redesign of class diagrams. An interpretation function is generated by a mapping satisfying conditions analogous to orthogonality in term rewriting systems. A similar concept was introduced in [Ko01], but it concerned the compositionality property only. Our concept of redesign is more general than the concept of refinement since we do not assume that properties are only added or refined, but they can be changed in an arbitrary way - for example, a number of design level classes might be restructured or a specification level class might be split into several design level classes. Properties which have to be preserved, may concern dependencies between classes, associations, operations, or generalization relationships. They are expressed in OCL, and transformed.

We show how a mapping defined on atomic model elements can be extended to complex model elements, and formalized as an interpretation function, if certain extendability conditions are satisfied. This function allows us to transform OCL specifications. The idea is that the designer or implementer who changes a class diagram maps the atomic elements on the target model elements, with the transformation of OCL-constraints being accomplished automatically.

Interestingly, our approach allows for not only an automatic constraints transformation but also for an automatic proof transformation. In the technical report [Ko05], we have shown that basic kinds of reasoning are preserved by interpretation functions; in particular proofs using propositional tautologies, resolution, and induction. This allows one to save on the clerical work of redoing proofs after transformation of class diagrams.

Interpretation functions allow us to trace model elements such as OCL-constraints through the software life cycle in both the forward, and the backward direction. In this paper, we

define a formal notion of forward, and backward trace, and show how to trace requirements throughout software development.

We illustrate our approach with a series of examples. In the first example, we transform a navigation path. In the second example, we describe a simple flip-flop game using state machines. In the second example, the states are implemented using enumeration types, and then they are implemented using objects instead of enumeration types. We discuss also the way one can abstract from irrelevant details. We discuss the applicability of our approach to the refactoring patterns of Fowler [Fo00], and identify the patterns which cannot be formalized using compositional functions.

The rest of the paper is structured as follows. Section 2 outlines our concept of redesign. Section 3 contains basic definitions. In section 4, we define the notion of interpretation function, and discuss its properties. Section 5 contains a series of examples which show how interpretation functions can be used to redesign class diagrams. In section 6, we discuss applications to refactoring patterns. In section 7, we show how interpretation functions can be used for tracing requirements. Section 8 is devoted to related work. Section 9 concludes the paper with remarks on the scope, and applicability of our approach.

2 The concept of redesign

Our approach is motivated by the concept of abstraction relationships as it is used in UML [UML05]. This notion allows us to relate model elements in different specifications. We are interested here in relationships modelling refinement, realization, or redesign of class structures. Such relationships can then be formalized, and used to automatically transform OCL-constraints.

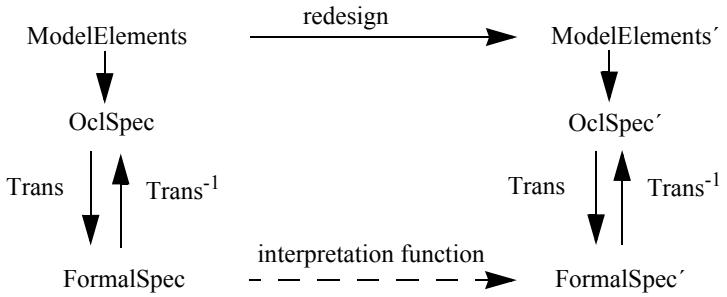

Fig. 1: Logical dependencies

Our concept generalizes the notion of interpretation function used by Taylor [Ta73]; the idea is to relate structures with the same behaviour but with possibly different signatures. The term ‘interpretation’ means that we interpret elements of one structure in terms of another. In our case, an interpretation function is a compositional function generated by a mapping that satisfies conditions analogous to orthogonal term rewriting systems (see [Te93]).

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 another. This is well modelled by partiality. To compare different specifications, selected model elements in the first specification are mapped to model elements in the second one. Such a mapping can be formalized in the first-order logic, and then extended to an interpretation function. We provide a sufficient condition to guarantee the existence of such functions. The idea of this approach is that a mapping has to be defined on simple model elements playing the role of 'bricks', i.e. regular elements which can be composed into more complex elements. Such a mapping can be then extended to the composed elements. The extension of such a mapping forms an interpretation function. Interpretation functions can be used to transform OCL-constraints, and the accompanying proofs.

Fig. 1 shows the relation between introduced concepts. Model elements are implicitly modelled by OCL terms. OCL-constraints are translated to first-order logic by the function Trans. Redesign of selected model elements generates an interpretation function on the level of OCL, and on the level of formal specifications, if extendability conditions are satisfied. In such a case this diagram commutes.

We do not assume that properties are added or refined, but we allow them to be changed in an arbitrary way as long as selected properties are kept unchanged. For example a number of design level classes might be restructured. With OCL we express system properties such as dependencies between classes, associations, operations, or generalization relationships.

Usually when performing redesign of a specification one has an intuitive idea what the trace of one specification in the other is. For example, if in consecutive specifications two equally named classes exist, then the class of the latter specification is assumed to implement or redesign the class of the former one; similarly for equally named operations.

In UML, a constraint is always attached to a UML model element called context. In particular OCL-constraints can be attached to classes in a class diagram. The syntactic correctness, or more precisely the type correctness, of an OCL formula depends on the context. Furthermore, changing the context may make a formula not only false but syntactically incorrect. In particular, tautological OCL formulas may become invalid. In the formal framework presented in this paper, the role of the context is played by the type system generated from a class diagram, and consequently the type correctness of a formula depends on the derived type system. Redesigning contexts requires rewriting corresponding OCL formulas. As the compositional functions are partial, they can be seen as a partial translation from one language to another. Those formulas which are not translated, should be considered as having been discarded. Accordingly, it is unreasonable to expect consequences of discarded formulas to be preserved, even if those consequences are translated. Interpretation functions are by definition generated by orthogonal mappings. The fundamental property of such functions is that under certain conditions they preserve reasoning using propositional tautologies, and resolution [Ko05]. On the other hand, the orthogonal-

ity guarantees that the translation of an OCL constraint is unique. Compositionality allows scaling up of a mapping defined on few model elements. Another important property of our approach is that it is constructive, in the sense that an interpretation function can be automatically generated from a dependency relationship, and used to transform constraints. The transformation can be automatized using term rewriting tools such as TOM [Ki+05]. There exist efficient unification algorithms (cf. [Te03]) that can be used for checking the orthogonality property.

We study the problem of requirements tracing in the case of complex UML class diagrams with OCL constraints, and we show how to use UML dependency relationships to trace constraints. We introduce UML stereotypes to mark appropriate dependency relationships. We formally define the notion of trace, which allows us to navigate through different variants of specification in the direction the software is developed (forward trace), in the backward direction (backward trace), and in both directions (full trace).

Our approach applies to specification on the level of class diagrams with constraints. We do not consider dynamic binding, but we do follow design by contract approach (see [Me98]), and we assume that subclasses inherit the constraints of their super-classes.

3 Basic definitions

In this section we show how to formalize some basic OCL expressions [UML05]. This semantics is a slight modification of the semantics defined in [BH+99]. The latter semantics is consistent with the semantics defined in [BF98] which is well integrated with the UML metamodel (see the related work section). More explanations, and in particular the underlying algebraic notions, can be found in the appendix.

The OCL types are modelled by sorts. In particular, types such as `Boolean`, `String`, `Integer`, and `Real` are modelled by sorts Boolean, String, Integer, Real, respectively[2]. OCL operations defined on such types are modelled by functions. For a sort A, the sorts Set(A), OrderedSet(A), Bag(A), and Sequence(A) are subsorts of the sort Collection(A). We define also a sort for class names ClN, and for each class name $C \in ClN$ we add equally named sort C of object identifiers. Id denotes the set of all object identifiers.

The sort Env models the environment. The elements of the sort Env correspond to the states of the heap. An environment can be also seen as a global state or a snapshot. An operation execution may involve the object possessing the operation as well as other objects. Therefore algebraic terms formalizing those operations have at least two arguments: an environment and an object name corresponding to the hidden parameter. OCL uses '.' to separate the implicit parameter `self` from an object's property or a method name. We use a similar notation in that the environment, and the implicit parameter are written on the left hand side of '.', whereas other arguments are written on the right hand side of the function symbol.

2. We use Courier font for OCL terms, and Times for algebraic terms as defined in the appendix.

The OCL expression `o.oclIsTypeOf(C)` means that object `o` is of class `C`, but not of any of its proper subclasses. It is formalized by the function

_ _ .oclIsTypeOf(_) : Env × Id × ClN → Boolean

We assume that (e, o).oclIsTypeOf(C) evaluates to true, if the object named o in the current environment e is of sort C, but not of any of its subclasses. Similarly, the OCL Boolean expression `o.oclIsKindOf(C)` states that `o` is of class `C` or of any of its subclasses; this operation is modelled by the function

_ _ .oclIsKindOf(_) : Env × Id × ClN → Boolean

An operation `op(`x_1 `:` T_1`,…,` x_n `:` T_n`)[:T]` may return a value of a type `T` and/or change the environment. If the operation is a query, then the environment component is kept unchanged. A query operation returning a value of type T is modelled by the function

_ _ .op _..._ : Env × C × T_1 ×…× T_n → T

We need the Env parameter since `op` may depend on the current state of the environment. An operation `op`, which changes the environment only is modelled by the function

_ _ .$op_{Env\_...\_}$: Env × C × T_1 ×…× T_n → Env

If an operation `op` returns a value, and changes the environment, then it is modelled by a pair of functions op, and op_{Env}. 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 formalize associations between classes. If `lnkB` is a directed association from class `A` to class `B`, then we formalize `lnkB` by the function

_ _.lnkB : Env × A → B,

if the association is single valued, or as

_ _.lnkB : Env × A → Set(B),

if the association has multiplicity larger than 1.

To formalize OCL constraints, we define the partial function Trans by structural induction. This function is defined on OCL terms containing variables, queries, attributes, and association-ends as well as on OCL predefined properties. The codomain of this function are algebraic terms as defined above. We do not treat OCL definitions, constraint names nor let-expressions here. The variable env of sort Env corresponds to the current environment where an OCL expression is evaluated. To translate invariants, and pre-conditions it is enough to translate OCL-terms which do not include `@pre` (in the case of formulas containing `@pre` we need a slightly more sophisticated definition, see subsection 5.1). The function is defined as follows:

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

Trans(`u.a`) $=_{def}$ (env, Trans(`u`)).a, for an OCL term `u`, and an OCL property `a`

We assume that Trans preserves the basic OCL operations such as addition, equality, and conjunction. An OCL term of the form `self.`a_1 … a_{n-1}`.`a_n is translated to the term of the form a_n(env, a_{n-1}(env,...a_1(env, self)...)) written in the standard prefix notation. On the

other hand, Trans has an inverse function $Trans^{-1}$ defined on formalizations of OCL terms, i.e. the codomain of Trans. Consequently, $Trans^{-1}(Trans(u)) = u$, for all OCL terms `u` such that Trans(`u`) is defined. The inverse function allows us to translate algebraic terms into OCL terms. For an OCL invariant of the form:

```
context C inv: Ψ
```

we obtain its formalization, if Trans(Ψ) is defined[3]:

$\forall_{env : Env,\ self_C : C}\ Trans(\Psi)$

3.1 Example

The idea of our approach is to relate different class diagrams by an appropriate dependency relationship. In this subsection, we show how to translate OCL formulas to first-order logic. We consider a contraction of a navigation path as it is defined by the *Remove Middleman* refactoring pattern (cf. [Fo00]). The path `lnkI.lnkB` shown on the left hand side of Fig. 2 is reduced to `lnkB` by deleting the intermediate class `I`.

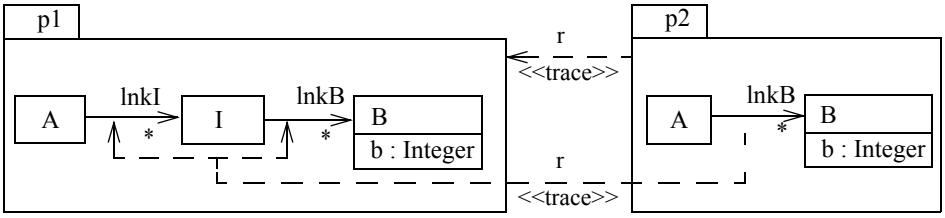

Fig. 2: Remove Middleman

Let us consider also the following two invariants:

```
context B inv bInv:
self.b <= 1

context A inv derInv:
self.lnkI.lnkB.b->sum() <= self.lnkI.lnkB->size()
```

The second invariant `derInv` can be derived from the invariant `bInv` by induction on the number of objects contained in `self.lnkI.lnkB`. If `self.lnkI.lnkB` is empty (contains no objects), then the formula

```
self.lnkI.lnkB.b->sum() <= self.lnkI.lnkB->size()
```

clearly holds. If this formula holds in the case when `self.lnkI.lnkB` contains n elements, then it holds when it contains n+1 elements. This is due to the fact that adding a new element increases the number of elements by 1, but the sum is increased by 0 or 1, depending on the value of the attribute `b` of the new element (see the appendix).

3. To avoid numeric indexes, we often extend the variable `self` by the first letter of the name of the corresponding class.

We formalize the class diagram on the left hand side, and the corresponding OCL constraints. We introduce sorts A, I, B for classes `A`, `I`, `B`, respectively. Associations `lnkI`, `lnkB`, and the attribute `b` are formalized by functions of the form:

$\_\_.\text{lnkI} : \text{Env} \times \text{A} \rightarrow \text{Set(I)}$
$\_\_.\text{lnkB} : \text{Env} \times \text{I} \rightarrow \text{Set(B)}$
$\_\_.\text{b} : \text{Env} \times \text{B} \rightarrow \text{Integer}$

The first invariant is formalized as follows:

$$\forall_{\text{env : Env, self}_\text{B} : \text{B}}(\text{env, self}_\text{B}).\text{b} \leq 1$$

The second invariant is formalized as follows (see the appendix):

$$\forall_{\text{env : Env, self}_\text{A} : \text{A}}\,(\text{env, (env, self}_\text{A}).\text{lnkI}).\text{lnkB.b->sum}() \leq |(\text{env, (env, self}_\text{A}).\text{lnkI}).\text{lnkB}|$$

4 Interpretation function

In this section we introduce the notion of interpretation function to formally relate class diagrams, and to transform the corresponding OCL-constraints. Moreover, we demonstrate how to derive interpretation functions from dependency relations. In subsection 4.1, we define the notions of orthogonal mappings, and compositional functions. A manual specification of a dependency relation can be very laborious - therefore, in subsection 4.2, we provide a method for extending relationships defined on a few model elements to complex dependency relations. In subsection 4.3, we define the notion of interpretation functions, and present two fundamental properties of those functions. We carry the Remove Middleman example through this section.

4.1 Formal definition

Dependency relationships are used to compare different specifications [UML05]; selected model elements in one specification are mapped to the related model elements in another one. In this subsection, we define a formal counterpart of this relationship. Model elements are formalized by terms, thus we obtain a mapping on terms. We consider here a simple version of the order-sorted algebra introduced by Goguen, and Meseguer (see [GM92]).

Term rewriting is a general model of computation. It has been successfully applied in many areas of computer science such as implementation of abstract data types, the foundations of functional programming, automated theorem proving, as well as code optimization, to name just a few (cf. [Te03]). One can see terms as building blocks that can be composed to form more complex structures. Term rewriting can be seen as a method allowing one to replace those blocks. Orthogonal term rewriting systems are the most regular ones. Roughly speaking, a set of term rewriting rules is orthogonal, if the rules do not overlap. In such a case, an application of one rule does not exclude an application of another. As a result, the rules can be applied in an arbitrary order. The applicability of a term rewriting rule does not depend on variable names, therefore here we do not treat variable

renaming explicitly (see [Ko05] for details). In this section we use the notion of algebraic terms, and signatures introduced in the appendix.

We say that two algebraic terms u, and v *overlap*, if at least one of the following conditions is satisfied:

- u and v are different and u can be unified with v after suitable variable renaming.
- u can be unified with a non-variable, proper subterm of v, or vice versa.

We say that a set of terms is *overlapping free*, if it does not contain terms which overlap. We say that a term t is *linear*, if for every variable x, t contains x at most once. We say that a set of terms is *orthogonal*, if it does not contain variables, it is overlapping free, and only contains linear terms. Orthogonality means that terms are like bricks, which can be fitted together and replaced independently. Note that the set $\{(env_1, (env_2, self_A).lnkI).lnkB, (env, self_B).b\}$ is orthogonal (see subsection 3.1).

Let $\psi : T(\Sigma, X, \tau) \rightarrow T(\Sigma', X', \tau')$ be a partial function. ψ is *compositional* iff for all terms t the following conditions hold:

i) $\psi(x)$ is defined for every variable $x \in X \cap X'$.

ii) If $\psi(t)$ is defined, σ is a variable renaming, then $\psi(t^\sigma) = \psi(t)^\sigma$.

iii) $var(\psi(t)) \subseteq var(t)$, if $\psi(t)$ is defined.

iv) If ψ maps term t_i to the term t_i', for i = 0,..., n, x_1,..., $x_n \in X \cap X'$ and term t has the form $t_0[t_1/x_1,..., t_n/x_n]$, then the substitution $t_0'[t_1'/x_1,..., t_n'/x_n]$ is well defined and $\psi(t)$ has the form $t_0'[t_1'/x_1,..., t_n'/x_n]$.

Conditions (i) and (ii) imply that compositional functions are defined on common variables and that they do not depend on variable's name. Condition (iii) is the standard requirement in the case of term rewriting systems (cf. e.g. [Te03]). (iv) is a compositionality condition; it allows us to scale up a mapping to complex terms. This condition can be equivalently expressed by a monotonicity condition of the form: $\tau(t_i) \leq \tau(x_i)$ implies that $\tau'(t_i') \leq' \tau'(x_i)$, for $x_i \in X \cap X'$. It is worth noting that the composition of compositional functions is a compositional function, provided that there is no conflict between variable names (see [Ko05]).

Let $A \subseteq T(S, F, \leq, X, \tau)$ be a set of terms. A mapping $\varphi : A \rightarrow T(S', F', \leq', X', \tau')$ is *orthogonal*, if there exists a partial function on sorts $\rho : S \rightarrow S'$ such that the following conditions are satisfied:

a) For every variable x, $\rho(\tau(x))$ is defined iff $x \in X \cap X'$.

b) If $x \in X \cap X'$, then $\rho(\tau(x)) = \tau'(x)$.

c) $var(\varphi(v)) \subseteq var(v)$, for $v \in A$.

d) If $v \in A$, then $\rho(\tau(v))$ is defined and $\rho(\tau(v)) = \tau'(\varphi(v))$.

e) If $\rho(s_1)$, $\rho(s_2)$ are defined and $s_1 \leq s_2$, then $\rho(s_1) \leq' \rho(s_2)$.

f) A (i.e. Dom(φ)) is orthogonal.

Conditions (a) and (b) say that the sort mapping ρ is determined by types of common variables; they are analogous to conditions (i) and (ii). Condition (c) is analogous to (iii). Condition (d) says that ρ commutes with φ with respect to types. Condition (e) says that ρ is monotone. Let us notice that in the case of a single-sorted algebra, any orthogonal term rewriting system determines an orthogonal mapping, and vice versa any orthogonal mapping determines an orthogonal term rewriting system.

To exemplify this definition, we define a sort mapping ρ_1 and a term mapping φ_1. Let ρ_1 map sorts A, B to A, B, respectively, and let φ_1 map the term $(env_1, (env_2, self_A).lnkI).lnkB$[4] to the term $(env_1, self_A).lnkB$. Since ρ_1 and φ_1 satisfy conditions (a),..., (f), in particular ρ_1 is monotone and the domain of φ_1 is orthogonal, the mapping φ_1 is orthogonal as well. Let us observe that in order to obtain orthogonal mappings, we need to deal with linear terms, i.e. terms which do not contain multiple occurrences of the same variable. In general, linearity does not allow one to map a term in different ways depending on whether some of its variables are different or not: it requires more general definitions of the mapping. On the other hand, queries do not change the state of the system, and env_1 is equal to env_2. In our case, the mapping of the linear term $(env_2, (env_1, self_A).lnkI).lnkB$ to the term $(env_1, self_A).lnkB$ implies that the non-linear term $(env_1, (env_1, self_A).lnkI).lnkB$, is mapped to the term $(env_1, self_A).lnkB$ by the generated interpretation function. Indeed due to compositionality and the fact that compositional functions preserve variables, $(env_1/env_2, (env_1, self_A).lnkI).lnkB$ is mapped on $(env_1, self_A).lnkB$.

In this paper we consider only compositional functions preserving predefined OCL types such as booleans or reals. A compositional function can be extended to boolean valued terms in the following way:

If $\varphi(\Phi) = \Phi'$ and $\varphi(\Psi) = \Psi'$, then $\varphi(\Phi \wedge \Psi) =_{def} \Phi' \wedge \Psi'$.

If $\varphi(\Phi) = \Phi'$ and $x \in X \cap X'$, then $\varphi(\forall_x \Phi) =_{def} \forall_x \Phi'$.

Similarly we can define the extension for other boolean connectors.

A compositional function can be applied to OCL terms. Given a compositional function φ, we can define the adjacent function $\phi(u) =_{def} Trans^{-1}(\varphi(Trans(u))$, for an OCL term `u` (cf. section 3). Let ρ map the sort C corresponding to class `C` to the sort C′ corresponding to class `C′`. An OCL constraint of the form

```
context C inv: Ψ
```

can be transformed to a new OCL constraint of the following form if and only if φ(Trans(Ψ)) is defined:

```
context C′ inv: ϕ(Ψ)
```

4. Note that the variables env_1 and env_2 occur at different positions in this term. Therefore to make the term linear we have to use different variables (see below).

4.2 Defining orthogonal mappings

Fully manual specification of a dependency relationship may be very laborious, in particular when the redesigned specification is very large. Fortunately, this process may be partially automatized. In this subsection we present a method for an automatic extension of dependency relationships, and a method for a systematic extraction of orthogonal mappings from dependency relationships [UML05]. A dependency relationship relates model elements such as packages, classes, attributes and operations. 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. Abstraction has three predefined stereotypes: «derive», «refine» and «trace». The derived abstraction specifies that the client may be computed from the supplier. The refine abstraction specifies refinement relationship between model elements at different levels, such as analysis and design. And finally, 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. The last two of these stereotypes are relevant to the redesign of class diagrams since, during redesign, a specification can be refined or traced. We use those stereotypes as additional benchmarking of dependency relationships without giving them any formal meaning.

To compare different specifications, selected model elements in the model being redesigned are related by a dependency relationship to the corresponding model elements in the target model.

An orthogonal mapping can be derived from a dependency relationship in three steps:

1) The dependency relationship is extended to packages, classifiers and properties.

2) The specification is formalized.

3) An orthogonal mapping is generated from the dependency relationship.

The usual way of redesigning a specification is to copy the specification to be modified and then to modify the copy. When a model element is not created, but copied, it is automatically linked to its ancestor. It is necessary to decide whether a model element must be coupled with an ancestor or not only when the model element is created or modified.

To facilitate mapping of model elements (see. step (1)), we use the rule that the dependency relationship is inherited by components of related model elements if they are of the same type and have the same names. More precisely, let a_1 and a_2 be model elements and let r be a dependency relationship. If the following conditions are satisfied:

- there exist model elements m_1, m_2 such that m_1 is mapped to m_2 by r
- for i = 1, 2, m_i is a package and m_i contains/imports model element a_i, or m_i is a classifier and m_i contains property a_i or m_i is a feature and a_i is its parameter
- a_1 and a_2 are equally named (i.e. $a_1.name = a_2.name$)

then r maps a_1 on a_2, unless a_1 is mapped to another element.

We propagate the linking from composed model elements to their parts starting with packages, through classifiers and constraints, to features and parameters. We relate packages with packages, classifiers with classifiers, properties with properties and operations with operations.

Using this procedure, we can extend the relation r defined in subsection 3.1 to classes A and B, since those classes are equally named in both packages. The extension maps class A in package p1 to class A in package p2, and class B in package p1 to class B in package p2. Furthermore we can extend it to the attribute b of class B, since this attribute occurs in both copies of class A, and since both copies are related. Note that the orthogonal mapping φ_1 defined in subsection 4.1 is a formalization of this extension.

To facilitate mapping of OCL terms (i.e. step (3)), we use the following rules:

- For every term t of a basic OCL type s (cf. [UML05]), the term $\rho(s) = s$, if $\varphi(t)$ is defined.
- If t has the form $f(x_1,..., x_n)$, f is a predefined OCL operation and the variables are of the basic OCL types, then $\varphi(t) = t$.

Since we want to obtain an orthogonal mapping, the completion procedure may add a new element to the domain of a constructed function only if it does not violate conditions (a),..., (f). In particular the added term must be linear and must not overlap with the formalization of the other model elements in the domain. Note that this can be checked automatically. In general the extension procedure can be fully automatized.

Let us consider the mapping φ_1 again. The second part of the extension procedure allows us to map the term |x| to the term |x| and the term y->sum() to the term y->sum(). Note that the set $\{(env_1, (env_2, self_A).lnkI).lnkB, (env, self_B).b, |x|, x->sum()\}$ is orthogonal. Consequently we can extend the orthogonal mapping φ_1 to an orthogonal mapping φ_2, which maps |x| to |x| and y->sum() to y->sum().

4.3 Properties of interpretation functions

Compositionality and orthogonality imply faithfulness of a translation. In this subsection we list two essential properties of interpretation functions. The first one guarantees that orthogonal mappings can be extended to compositional functions. This property is proved in the appendix. The second property states that interpretation functions preserve certain kinds of proofs. We refer the interested reader to [Ko05] for the proof of the second property.

Extendability Theorem

Let $A \subseteq T(\Sigma, X, \tau)$ be a set of terms and let $\varphi : A \to T(\Sigma', X', \tau')$ be an orthogonal mapping. Then the mapping φ can be uniquely extended to a compositional function

$\psi : T(\Sigma, X, \tau) \rightarrow T(\Sigma', X', \tau')$

such that Dom(ψ) is the smallest set containing A, $X \cap X'$ and closed on term composition.

A compositional function is an interpretation function, if it is generated by an orthogonal mapping: ψ is *interpretation function* if, and only if, there exists an orthogonal mapping φ such that $\varphi(t) = \psi(t)$, for every term $t \in Dom(\varphi)$, and Dom(ψ) is the smallest set containing the domain of φ, $X \cap X'$, and is closed on term composition. Below we will strictly distinguish between mappings, and functions generated by mappings.

Let us consider the mapping φ_2 defined in subsection 4.2. We cannot apply this mapping to terms such as (env, (env, $self_A$).lnkI).lnkB.b->sum() and |(env, (env, $self_A$).lnkI).lnkB|, since those terms do not belong to the domain of this mapping. Therefore we need to extend this mapping to complex terms such as the two above. The domain of φ_2 is orthogonal and ρ is monotone. Therefore φ_2 induces an interpretation function. This function applies to both terms. Consequently, we can transform the formulas defined in subsection 3.1. The formalization of the first invariant is transformed by the resulting interpretation function into the following formula:

$\forall_{env : Env,\ self_B : B}(env, self_B).b \leq 1$

The formalization of the second invariant is transformed as follows:

$\forall_{env : Env,\ self_A : A}\ (env, self_A).lnkB.b->sum() \leq |(env, self_A).lnkB|$

The formulas above correspond to the following OCL-constraints (cf. subsection 4.1):

```
context B inv bInvT:
self.b <= 1

context A inv derInvT:
self.lnkB.b->sum() <= self.lnkB->size()
```

The results presented in [Ko05] show that interpretation functions preserve reasoning using resolution, modus ponens, propositional tautologies, and induction. A proof using resolution, modus ponens and propositional tautologies is called a *PTR-proof*. Let the sort s be generated by a constant constructor c and a unary constructor f[5]. An inductive proof of a formula of the form $\forall_{x : s}\ \Phi(x : s)$ has two parts:

- A proof of $\Phi(c)$, for a constant constructor c.
- A proof of $\Phi(x) \Rightarrow \Phi(f(x : s, y_1 : s_1,.., y_n : s_n))$, for a constructor f.

---

5. For simplicity we assume that c and f are the only constructors of sort s. We assume also that the set A_S corresponding to the sort s is the smallest set such that $c^A \in A_S$, and such that if $a \in A_S$, $a_1 \in A_{s1},..., a_n \in A_{sn}$, then $f^A(a, a_1,..., a_n) \in A_S$ (see the appendix).

Proof Preservation Theorem

Let $\psi : T(S, F, \leq, X, \tau) \rightarrow T(S', F', \leq', X', \tau')$ be an interpretation function. Let S have a tree structure and let the underlying sort mapping ρ map the largest sort of S to the largest sort of S′. If ψ is defined on a set of formulas Ax and on the formula Φ, then the following holds:

- If there is a PTR-proof of the formula Φ using Ax, then there is a PTR-proof of $\psi(A)$ using $\psi(Ax)$.
- Let us assume that the formula Φ has one free variable x. If there is a PTR-proof of $\Phi(c)$, and if there is a PTR-proof of $\Phi(x) \Rightarrow \Phi(f(x, y_1,..., y_n))$, then there is an inductive proof of Φ preserved by ψ.

The assumption that ρ preserves the largest sort corresponds to the requirement that the OCL type `OclAny` (see [UML05]) is preserved. In the paper [Ko05] we show that this theorem holds under a less strict condition. The assumption that the sorts form a tree structure corresponds to the requirement that there is no multiple inheritance. It is worth mentioning that resolution and propositional tautologies are the standard means of reasoning in first order logic and that other reasoning rules are more difficult to use. The fact that interpretation functions preserve PTR-proofs has several advantages. Interpretation functions can save the often tedious work of redoing proofs after class structure redesign. In particular, if there is a proof that an invariant implies a pre-condition and that the corresponding post-condition implies the invariant, then the same relation holds for the transformed formulas, provided that the proof uses the above-mentioned ways of reasoning. Similarly, if one invariant implies another invariant, then this relation is preserved by an interpretation function. For example in the case of the Remove Middlemen refactoring, the Proof Preservation Theorem implies that `bInvT` implies `derInvT`. Consequently, it is not necessary to redo the proof.

The inverse mapping corresponds to the *Introduce Middleman* refactoring [Fo00]. This mapping can be extended to an interpretation function as well. The Remove Middleman example demonstrates that a partial function is really needed since we neither map `lnkI` nor `lnkB`. Let us observe that the interpretation function as defined in [Ta73] does not suffice, since we map complex terms such as $(env_1, (env_2, self_A).lnkI).lnkB$, not only single function symbols. For the same reason, we cannot apply the forget functor to signatures in order to remove `lnkI` and `lnkB`, since we would remove `lnkI.lnkB` as well.

5 Examples

During the software engineering process a class structure can be amended several times and a variety of design and refactoring patterns can be applied [GH+95, Fo00]. In this section we show how to use our technique in the case of class structure redesign. Interestingly enough, the mappings considered are orthogonal and generate interpretation functions.

5.1 Transforming state machines

This subsection addresses the problem of relating different implementations of state machines. State machines are one of the basic means to describe object behaviour in UML. Object-oriented modelling allows for different implementations of state machines. For example, states in a state machine can be implemented by enumeration types or by state classes, as in the State Pattern [GH+95]. The problem arises as to how we compare such seemingly different implementations.

First, we implement states using an enumeration type and then using classes. The operations triggering transitions are constrained by pre- and post- conditions. Pre- and post-conditions can be formalized in a similar way to invariants, but one has to take care of different system states before and after execution of the operation. Let us consider an OCL constraint of the form:

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

[`:T`] means that the return type is optional. Similarly, the precondition is optional.

To translate the pre-condition, we use the function Trans as it is defined in section 3. To translate the post-condition we use the function $Trans_{post}$ [BH+99], which is defined like Trans, but with two different variables, env and env´, in the formalization:

- $Trans_{post}$(`t.a`) = (env´, $Trans_{post}$(`t`)).a
- $Trans_{post}$(`t.a@pre`) = (env, $Trans_{post}$(`t`)).a

The variables env and env´ added by function $Trans_{post}$ are of sort Env and model the environment before and after the execution of the operation `op`, respectively. The translation of the above OCL constraint has the form:

$$\forall_{\text{self : C, env, env´ : Env, x1 : T1,…, xn : Tn}} \text{env´} = (\text{env, self}).\text{op}_{\text{Env}}(x_1,\ldots, x_n) \wedge$$
$$\text{Trans}(\Phi) \Rightarrow [\forall_{\text{result : T}} \text{result} = (\text{env, self}).\text{op}(x_1,\ldots, x_n) \Rightarrow] \text{Trans}_{\text{post}}(\Psi)$$

Square brackets indicate that the part corresponding to `result` is optional. It occurs if `op` returns a value.

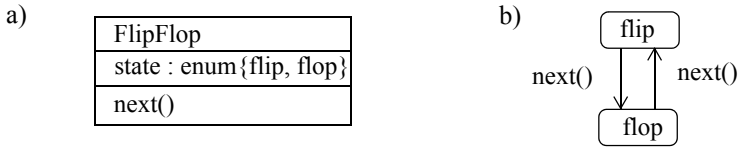

Fig. 3: FlipFlop game

Fig. 3 (a) shows class `FlipFlop`. The corresponding state machine is shown on Fig. 3 (b). An object of that class can be in the state `flip` or `flop`. There exists an operation

`next()` which transposes these states. The behaviour of this state machine can be specified in OCL as follows:

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

In OCL, the values of associations and properties in general can be treated as sets, therefore we can write `state.includes(#flop)` meaning that the value of `state` is `flop`. Formally, the definition of translation function does not allow us to treat constraint names explicitly, but the translation may be performed as if there were no names. The constraint above is formalized in the following way:

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

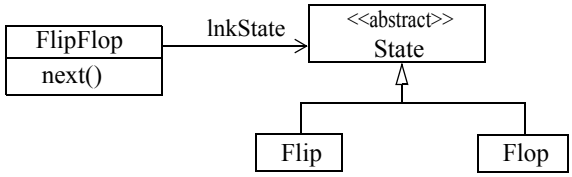

Fig. 4: States as classes

We may redesign this state machine using the State Pattern [GH+95]. In the redesigned version, the states are implemented by objects instantiating classes `Flip` and `Flop`, which extend the class `State` (see Fig. 4).

We map the elements of the enumeration type to the corresponding classes; i.e. `#flip` and `#flop` are mapped to classes `Flip` and `Flop`, respectively. Moreover, the OCL term `state.includes` is mapped to the term `lnkState.oclIsKindOf`. It is not difficult to observe that this mapping, defined on the level of model elements, induces an interpretation function. The transformed formula has the form:

$$\forall_{\text{self : FlipFlop, env, env' : Env}}\ \text{env}' = (\text{env}, \text{self}).\text{next}_{\text{Env}} \Rightarrow$$
$$((\text{env}, (\text{env}, \text{self}).\text{lnkState}).\text{oclIsKindOf}(\text{Flip}) \Rightarrow$$
$$(\text{env}', (\text{env}', \text{self}).\text{lnkState}).\text{oclIsKindOf}(\text{Flop}))$$
$$((\text{env}, (\text{env}, \text{self}).\text{lnkState}).\text{oclIsKindOf}(\text{Flop}) \Rightarrow$$
$$(\text{env}', (\text{env}', \text{self}).\text{lnkState}).\text{oclIsKindOf}(\text{Flip}))$$

Let us observe that the formalization of `lnkState.oclIsKindOf` is nonlinear. This does not contradict orthogonality of the corresponding mapping, since the definition re-

quires only the terms in its domain to be orthogonal and does not restrict the range of the mapping.

The formula above formalizes the following OCL constraint:

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

5.2 Abstraction

Class diagrams can be very complex and hard to read if they are very extensive and detailed. Egyed proposed a powerful method of abstracting away class diagram details, if they are irrelevant from a particular point of view [Eg03]. He proposed two basic types of abstraction: Compositional abstraction allows one to group multiple classes and relational abstraction allows one to contract paths in a similar way as it is done in the Remove Middleman and Inline Class refactoring patterns (see subsections 3.1). Egyed introduced 121 rules allowing one to deal with different kinds of relational abstractions. Those rules correspond to composition of different kinds of association (i.e. aggregation and composition) and inheritance. Interestingly in our formal framework, it is possible to deal with those rules in a formal and uniform way.

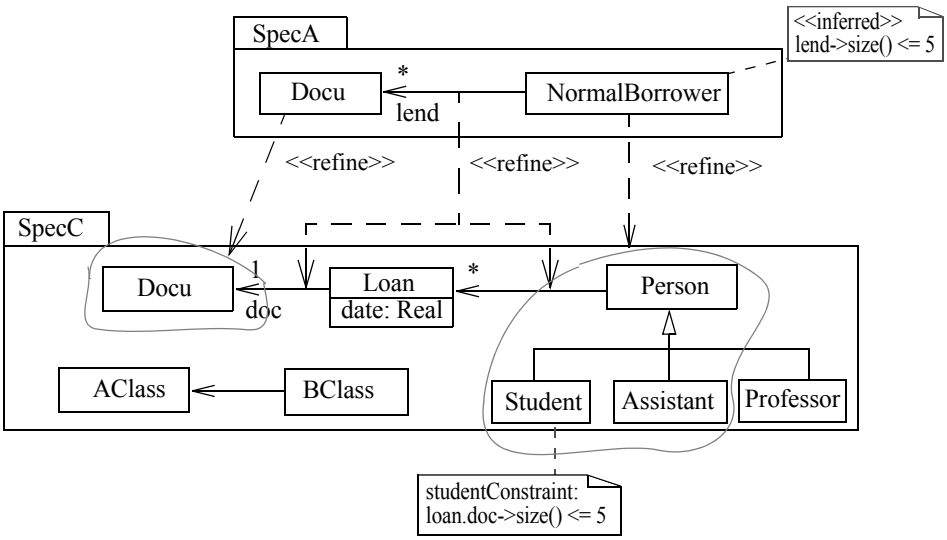

Fig. 5: Abstraction

This example shows how to deal with abstraction within our framework. In this subsection we consider a library which stores documents (see Fig. 5). There are three types of users: student, assistant and professor. The view specified in `SpecA` abstracts away from several details of the concrete view specified in `SpecC`. It groups classes `Person`, `Student` and `Assistant` into one class `NormalBorrower` (compositional abstraction in the terms of Egyed) and suppresses the classes `Professor`, `AClass` and `BClass`. We use the abstraction relationship `ar` to relate the abstract and the concrete view. We use the UML stereotype `<<refine>>` which allows us to relate different abstraction levels such as analysis and design (see [UML05]). Dependency relationships annotated with this stereotype indicate a transformation from the abstract to the concrete model, therefore the direction of arrows is opposite to the direction in the case of a trace relationship.

The formalization of this example is straightforward. We define the sort function ρ by mapping sorts corresponding to classes `Person`, `Student` and `Assistant` to the sort corresponding to `NormalBorrower`. We also map the term corresponding to the path `loan.doc` to the term corresponding to the association-end `lend`. This term mapping can be extended to an interpretation function and used to transform constraints. We introduce the stereotype `<<inferred>>` to indicate that a stereotyped constraint was inferred from another constraint.

As mentioned above, our formal framework allows us to study the rules introduced by Egyed in formal terms. There is a rule allowing one to compose two associations (like in the example above). There is a rule allowing one to compose an association and a composition, or a rule which allows one to restrict an association to a subclass (e.g. association `loan` can be restricted to the class `Student`). Most of them are just special cases of the interpretation function. Some of them, when formalized literally, violate the compositionality requirement. The author deals with this problem by putting reliability rating, but in our opinion this problem requires a formal treatment. Interestingly for some of those rules it is possible to formalize the associations using OCL terms in such a way that the compositionality is preserved. Our approach does not distinguish between formalization of an association in general, an aggregation and a composition. But we may express the corresponding semantics in OCL and deal with the resulting constraints.

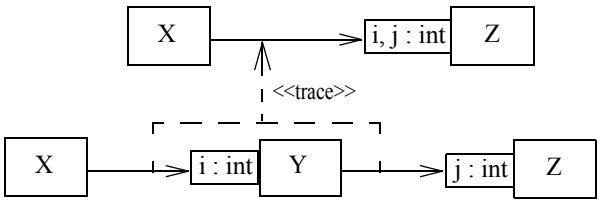

Fig. 6: Qualified association

In our opinion, any finite number of such rules does not suffice. It is not difficult to show that there are infinitely many ways in which OCL properties can be composed. For example, one would need infinitely many rules for qualified associations. Fig. 6 shows an example of class contraction.

This construction allows replacement of two qualified associations by one association with two qualifiers. Of course in the same way one can abstract from any number of qualified associations, but this requires an infinite number of rules. In general, there are infinitely many patterns of composition and only term composition can describe all of them.

6 Refactoring patterns

In this section we discuss the applicability of our method to the refactoring patterns of Fowler [Fo00]. We group the refactoring patterns into three categories: refactorings which introduce or remove model elements, refactorings which move or modify model elements, and refactorings which generalize or narrow types. We explain how our approach applies to each category. Let us point out that in this paper we consider only class structure refactoring, consequently we do not deal with the code refactoring.

Our approach allows us to formalize most refactoring patterns. We can use compositional functions when the refactorings are considered in isolation. Nevertheless, we have to distinguish between modification of a class diagram and a modification of its clients. In the case when a client relies on a particular signature of a class structure, changes of the class structure may break the client in the sense that the client may rely on functionality which is no longer available. In such a case the client has to be modified as well.

6.1 Introducing and removing model elements

We can deal easily with adding or removing model elements such as associations and operations, since the compositional functions can be partial and do not have to be surjective. Due to partiality we can formalize refactorings such as Changing Bidirectional Association to Unidirectional. Vice versa, we can formalize Changing Unidirectional Association to Bidirectional because of the fact that compositional functions do not have to be surjective.

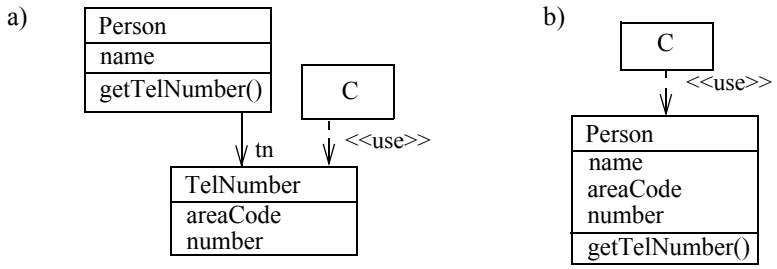

Fig. 7: Inline Class

Fig. 7 shows the Inline Class refactoring (we omit the <<trace>> relationships). In subsection 3.1, we have already considered a special case of this pattern. We use the stereotype <<use>> to indicate that the class C uses properties of the class TelNumber. If there are no clients using the removed class, then compositionality is preserved; in the other case a client can be broken. For example, let us suppose that the class Person is constrained by the following OCL-invariants (cf. Fig. 7 (a)):

```
context Person inv:
self.getTelNumber() = self.tn.number

context C inv:
self.telNumber.number >= 100000
```

We map tn.number to number, areaCode to areaCode and getTelNumber() to getTelNumber(). Formally, we define the function ρ by mapping sorts Person and TelNumber to the sort Person. We map also:

(env, $self_T$).areaCode to (env, $self_P$).areaCode

(env_1, (env_2, $self_P$).tn).number to (env_1, $self_P$).number

(env, $self_P$).getTelNumber() to (env, $self_P$).getTelNumber()

(env, $self_P$).name to (env, $self_P$).name

Note that we did not map sort C, corresponding to the client. The term mapping defined above (let us call it φ) can be extended to a compositional function ψ. Note that in the first constraint, the query on the left side of the equation is preserved by the mapping and that term on the right side is contracted. Consequently, this constrain is transformed to:

```
context Person inv:
self.getTelNumber() = self.number
```

ψ does not allow us to transform the second constraint, since it does not apply to class C. Therefore, we extend φ to φ_1 by mapping the term (env, $self_C$).telNumber to the term (env, $self_C$).person. The interpretation function generated by φ_1 allows us to transform the second constraint. The transformed constraint has the form:

```
context C inv:
self.person.number >= 100000
```

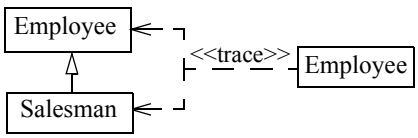

Fig. 8: Collapse Hierarchy

The Collapsing a Hierarchy refactoring merges a superclass with its subclass when they are similar (see Fig. 8). We use the stereotype <<trace>> to describe evolution of model

elements. In our case, the sort corresponding to the collapsed class is mapped to the sort corresponding to the upper class; this sort mapping preserves monotonicity. We map types `Employee` and `Salesman` to the type `Employee`. Those methods having an implicit parameter of type `Salesman` can be transformed to methods having implicit parameter of type `Employee`. Similarly, if a method returns objects of type `Salesman`, then it is transformed to a method returning objects of type `Employee`. This broadens the type of returned values, but it is in accordance with condition b), since both types are mapped to the same type. Consequently, this refactoring does not break clients of class `Employee`. The clients of `Salesman` have to use the `Employee` class henceforward.

The Introduce Middleman refactoring reverses the Remove Middleman refactoring. It creates a new class, when one class is doing work that should be done by two, and moves the relevant fields and methods to the new class. The clients using the functionality moved to the new class can access it via the old class that forwards the calls to the extracted class. This refactoring can be dealt with easily because of the fact that we allow mapping of a single function symbol to a complex term.

We can easily deal with removal of parameters. In Remove Parameter refactoring, a parameter is removed when it is no longer needed. This corresponds to skipping a variable in a function, e.g. f(x, y) can be mapped to f(x); this is in accordance with the condition that $var(\varphi(t)) \subseteq var(t)$, for a compositional function φ.

Introduce Parameter Object replaces a group of parameters that naturally go together by a single object. It can hardly be dealt with in the current state of our approach.

The Add Parameter refactoring adds a new parameter that can pass on some additional information. Similarly the Parameterize Method refactoring introduces a new parameter to a method. We cannot formalize these refactorings since by definition a compositional function does not introduce new variables (this assumption is usually made in term rewriting systems to exclude some pathological cases). Similarly we can hardly deal with the Encapsulate Collection refactoring, since it introduces new variables and the definition of the new operations cannot be formulated using term composition only.

6.2 Moving and modifying model elements

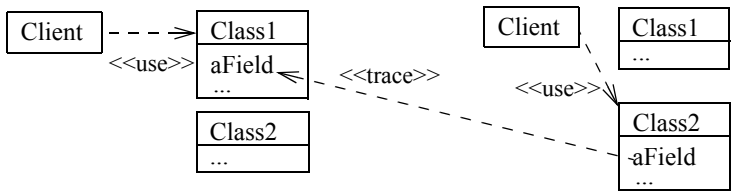

Fig. 9: Move Field

Refactorings such as Move Field or Move Method can be treated in our approach using compositional functions. Move Field creates a new attribute with a similar signature in the

class which the client uses most. The old attribute is removed altogether or turned into a simple delegation. This refactoring changes the type of the implicit parameter of the moved attribute which may result in a loss of compositionality. In this case we adjust the client (see Fig. 9).

The Hide Delegate refactoring creates methods on the server to hide the delegate. It can be dealt with by compositional functions, if the delegate class is not accessible to the clients. If it is accessible, then its clients have to be adjusted as in the case of Move Method.

The Pull up Method refactoring moves methods to the superclass, if they have identical results on subclasses. This refactoring can be dealt by compositional functions, since it broadens the types of the implicit parameter of the Moved Method.

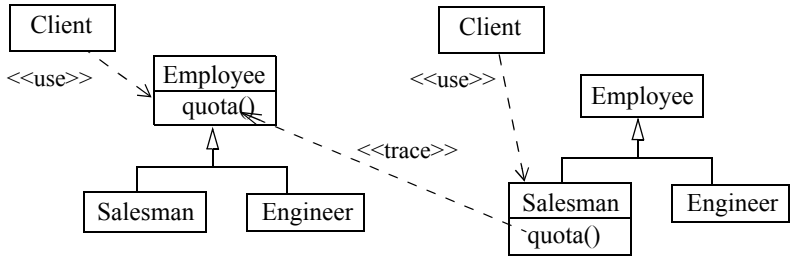

Fig. 10: Push Down Method

The Push Down Method refactoring is the opposite of the previous one. It may break compositionality and cannot be dealt with using compositional functions. After pushing down the methods their clients have to be adjusted (see Fig. 10).

The Replace Type Code with Subclasses refactoring replaces numeric type code attributes with a new class. In this case, the clients of the attributes must be transformed; for example `self.salesman > 0` may be replaced by `self.oclIsKindOf(Salesman)`.

6.3 Type generalization and narrowing

The Form Template Method refactoring pushes common methods from subclasses to the superclass. This refactoring can be treated by a compositional function since the generalization of the implicit parameter type does not break compositionality. It can be dealt with as Push up Method.

The Replace Delegation with Inheritance refactoring makes the delegating class a subclass of the delegate. Since the delegating class inherits operations and attributes from the superclass, this refactoring does not break clients, although the methods and constraints of the delegating class have to be adjusted. This refactoring is in accordance with compositionality conditions.

The Replace Inheritance with Delegation refactoring creates an association for the superclass, adjusts methods to delegate to the superclass, and removes the subclassing. This refactoring may break compositionality, since it breaks the subtype relation. Nevertheless, it can be dealt with by a sequence of compositional functions, adjusting the clients, so that the composition of these functions forms a compositional function.

7 Tracing requirements

Tracing requirements from the initial requirements specification to the final implementation is crucial for gaining control over the development process, it provides a level of project control and quality that would be very hard to achieve by any other means: You can't manage what you can't trace [WN94]. One of the greatest motivating factors for tracing requirements is that contracts require companies to do so [DD01]. It is usually practiced by software providers of high-reliability products and systems. A requirement is a software capability which, by definition, must be met by a system or component to satisfy a contract, specification, standard or other formally imposed document. Requirement's traceability is the ability to describe and follow the life of a requirement, in both a forward and backward direction, throughout the system life cycle [Ja98]. Since traceability has to operate in both directions, for any requirement it should be possible to trace its initial source as well as the corresponding test cases and the ultimate implementation.

The problem of traceability has been widely studied and many approaches have been developed. The paper [KP01] presents a taxonomy of different approaches. Tracing can be done at different levels of granularity and abstraction. Tracing from one document to another is the most basic. The most refined level provides the possibility to trace every single statement (cf. [KP01]). Interpretation functions provide refined traceability. The notion of tracing is based on the notion of relationship between traced entities. The relationship can relate software artifacts at the same and at different abstraction levels (see section 5). In UML, the corresponding notion is the notion of dependency relationship. The relation can be set in an implicit and an explicit way. The subsection 4.2 describes a combination of both.

7.1 Traces

During the software engineering process a class diagram may undergo several changes because of pattern applications, refactorings, refinements, etc. The model elements contained in such a diagram evolve over time. A trace of a model element is an ordered set of model elements which are antecedent or subsequent versions of the model element. Model elements are related by the dependency relationships. On the formal level, algebraic terms formalizing those model elements are related by compositional functions.

For a relation R, R^{-1} denotes the inverse of R and R* denotes the reflexive and transitive closure of R, i.e. $R^* = R^0 \cup R^1 \cup R^2 \cup ...$

Let F_0 be a set of compositional functions, we define the relation

$F =_{def} \{(t, t') \mid \exists_{f \in F_0} f(t) = t'\}$

Let U be a set of algebraic terms.

- The *forward trace* of U equals F*(U)
- The *backward trace* of U equals $(F^*)^{-1}(U)$
- The *full trace* of U equals $(F)^*(U) \cup (F^{-1})^*(U)$

The relation F* provides a quasi-ordering that corresponds to the evolution of model elements during the software development process. It allows for forward and backward navigation through different versions of specification and for tracing constraints. The forward trace of a set of model elements U is the image of U with respect to F*. It models the forward evolution of terms (at the formal level) and the corresponding model elements (at the UML level, see the next subsection). Similarly, the backward-trace models the history of U. The full-trace models the full lifeline of U, i.e. the history of U and its latter forms.

7.2 Example

In this subsection we reconsider the library example (see subsection 5.2). The library's borrowing and circulation policies depend on the type of the user. We show how an abstract specification of the library can be refined and modified in a series of steps. This approach can be also seen as view integration (cf. [Fi+94]), in which a design is obtained by combining different views.

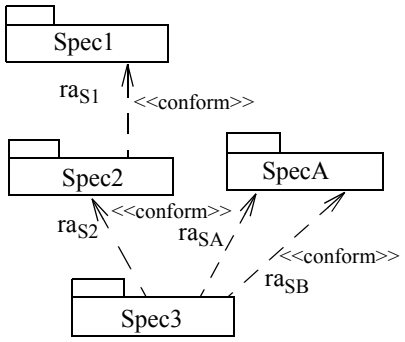

Fig. 11: Abstract dependencies

Fig. 11 shows a brief structure of this specification. To avoid confusion, in the following we name relationships, if there is more than one relationship of a given type. The packages `Spec1`, `Spec2`, `SpecA` and `Spec3` are related by stereotyped dependency relationships. We use the stereotype `<<conform>>` to indicate that model elements in one specification trace model elements in another specification and conform to their constraints. Let us recall that in UML the trace relationship is a particular kind of the abstraction relationship (see [UML05]).

Fig. 12 shows the detailed structure of the specification. We use package names to resolve name clashes when necessary. The trace relationships r_1, r_2, r_A, r_B are detailed version of the dependency relationships ra_{S1}, ra_{S2}, ra_{SA}, ra_{SB}, respectively. We put an arrow corresponding to a dependency relation in the background if it does not concern a sub-diagram.

`Spec1` is a rough specification. The class `Borrower` can indirectly reference objects of the `LoanableDocu` class via the `Junks` class. Specification `Spec1` is converted to `Spec2` by deleting the class `Junks` and introducing the association `borrow` for the path `junks.loan`. Packages `Spec1` and `Spec2` are related by the dependency relationship r_1. Therefore, thanks to the extension procedure (see step (1) in subsection 4.2), the class `Spec1::LoanableDocu` and the class `Spec1::Borrower` are related by r_1 with the class `Spec2::LoanableDocu`, and the class `Spec2::Borrower`, respectively. The only thing that the user has to do in this case, is to relate the association `junks.loan` and the association `borrow`.

The specification `SpecA` limits to 5 the number of borrowable documents a normal borrower can borrow (this is expressed by the `lendConstraint`). `Spec3` refines `SpecA` and `Spec2`. The dependency r_2 relates the associations named `borrow`, the class `LoanableDocu` and the class `Borrower` from the package `Spec2` with the composed association `loan.doc`, the class `Docu` and the class `Person` from `Spec3`, respectively. The dependency relationship r_A relates the class `Student` with the class `NormalBorrower` and the path `loan.doc` with the association `lend`. Similarly for r_B.

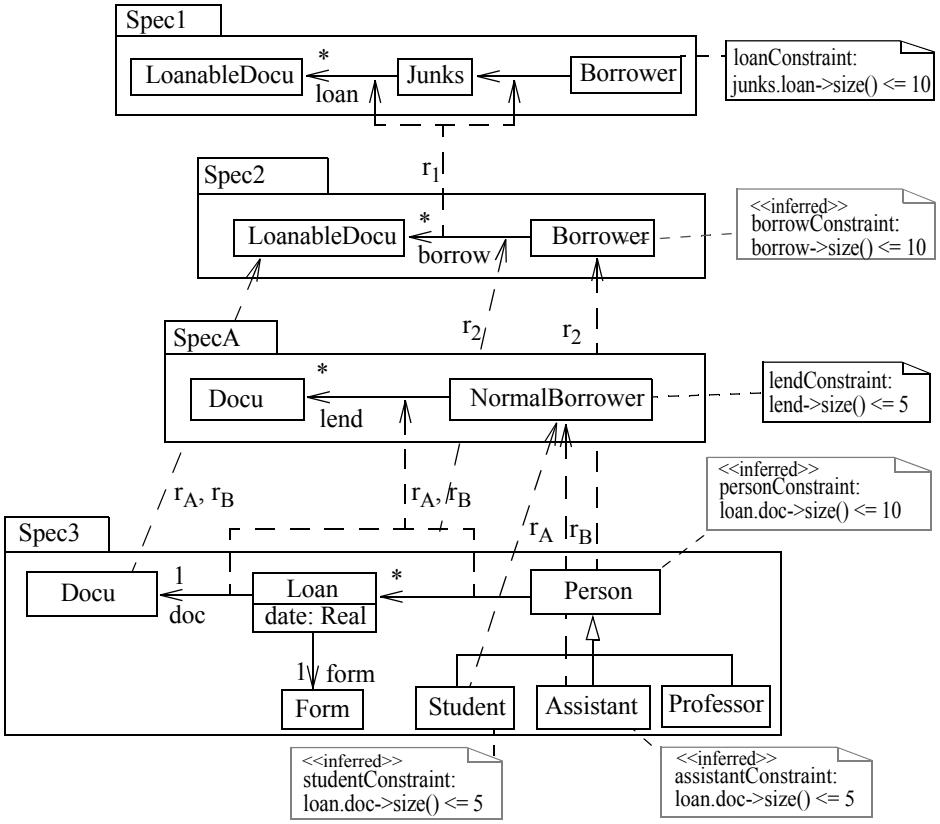

Fig. 12: Dependences

7.3 Example continued: Transforming constraints

In this subsection we show how to transform those constraints according to the dependency relationships (see Fig. 12). We show also how to use those dependency relationships and the corresponding interpretation functions to define traces.

Four compositional functions exist; they are φ_1, φ_2, φ_A and φ_B, which correspond to the dependency relationships r_1, r_2, r_{SA} and r_{SB}, respectively (in fact, these are interpretation functions). The dependency relationships in this structured specification impose interesting relations between different constraints.

The constraint in the `Spec1` package is expressed in OCL as follows

```
context Spec1::Borrower inv loanConstraint:
self.loan->size() <= 10
```

`Spec2` conforms to `Spec1`, therefore `loanConstraint` has to be to transformed using the generated compositional function φ_1. The resulting constraint has the form:

```
context Spec2::Borrower inv borrowConstraint:
self.borrow->size() <= 10
```

Consequently, an object of the class `Spec2::Borrower` can be related to at most 10 objects of the class `LoanableDocu`. We use the stereotype `<<inferred>>` to indicate that a constraint was inferred.

Since `Spec3` conforms to `Spec2` and the class `Spec3::Person` depends on the class `Spec2::Borrower`, the `borrowConstraint` can be further transformed using φ_2 to

```
context Spec3::Person inv personConstraint:
self.loan.doc->size() <= 10
```

We define traces using the set $F_0 = \{\varphi_1, \varphi_2, \varphi_A, \varphi_B\}$. To obtain the forward-trace of the `Spec1::junks.loan` association, we apply φ_1 and φ_2, and so get {`junks.loan`, `borrow`, `loan.doc`}. The forward-trace of the association `Spec1::borrow` equals {`borrow`, `loan.doc`}, the backward trace equals {`junks.loan`, `borrow`}, and the full trace equals {`junks.loan`, `borrow`, `loan.doc`}.

8 Related work

There exist a number of formal approaches to refactoring. Unfortunately, the research on formal foundations of class structure redesign, in particular on constraints transformation, is still underdeveloped. The interpretation function used in abstract algebra [Ta73] is defined for single sorted algebras specified by equations. It transforms a single operation into a complex term. Interpretation functions defined here can be seen as a generalization of this idea. The approach of Taylor [Ta73] does not allow to handle patterns such as Remove Middleman. The Lano's approach [La96] uses interpretation functions in the sense of Taylor and equational specification to define the notion of refinement for the Real-Time Action Logic; it is based on the Object Calculus [FiMa91]. This approach does not allow for a constructive transformation of constraints because of undecidability of eqational theories. Graph rewriting systems may be used to describe transformation of a specification (cf. e.g. [Gr99]), though they cannot be directly applied to transform or compare constraints. The approach of [Fra+02] uses the Role Pattern (cf. e.g. [Bä00]) to define refactorings of static structural UML models composed of classes and interfaces. A source model is transformed to a target model using a set of informally specified transformations.

In general, it is also possible to define compositional functions using equality and the forget, restrict and identify functors [Wi90], however this complicates the definition and makes the approach undecidable. The use of signature morphisms is also not general enough since it does not allow mapping complex terms on complex terms. In the example of Remove Middleman, we show that the forget functor cannot be used and that an interpretation function, in the sense of Taylor, does not suffice either. Our notion of interpretation function is very flexible; we show that it allows the treatment of non-incremental changes of class diagrams.

In the Catalysis book [DW98], the authors consider a redesign example which matches very well with our ideas. In this example a manager tries to figure out the trace of certain model elements. This example can be treated within our framework in a systematic way. The traced model element can be seen as the domain of an orthogonal mapping. Our approach allows us to generate an interpretation function from this mapping and to transform OCL-constraints (though no constraints were considered in that example). It should be pointed out that, of course, we are not the first to consider restructuring UML specifications with OCL-constraints, but to the best of our knowledge we are the first to do it in a systematic and formal way.

The view points framework [Fin+94] provides a systematic, logic-based approach to inconsistency handling. This is done not by eradicating inconsistencies, but by supplying logical rules specifying how to act on them. The rules are expressed on the meta-level using temporal logic. Strictly speaking, this approach is not aimed at redesign, but handling inconsistency. It uses also very powerful formal machinery, but is rather nonconstructive.

The current approaches to inconsistency usually embed models into another model or language (it is often first-order logic) and perform constraint-based reasoning in context of that model (see [Hu+97] and the references there). These approaches have shown interesting results. But there are two major problems. On the one hand, they lack change propagation, i.e. support for the subsequent, necessary adaptation of models once inconsistencies are found. On the other hand, the use of equations results in undecidability. Those approaches are also not meant for transforming constraints.

In the paper [ZMK05], the notion of specification redesign is defined in the terms of arbitrary institutions. Basic properties of redesign are investigated and the formalism is applied to provide a formal, institution-independent semantics for UML class diagram transformations. Refactoring patterns are described in terms of class diagrams and interpreted as redesigns of corresponding algebraic specifications. Interpretation functions can be seen as the constructive counterpart of this approach. This approach can hardly deal with refactorings such as Remove Middleman [Fo00] and provides conditions for traceability and redesign which are in general undecidable.

We consider here order-sorted algebras as introduced by Goguen and Meseguer (see [GM92]). The notion of order-sorted algebras have been extended to so-called membership equational logic [Bo+00], which is more powerful and flexible. There is a number of formal semantics of UML class diagrams. The paper [BF98] proposes such a semantics which fits well the UML metamodel (see also [Eva+99]). In this semantics, classes are formalized by sets of object references. There exists a valuation, which maps each reference to a set of attribute values. Generalization is modelled by the set inclusion relation. Set theoretic formulas are used to formalize the meaning of direct and indirect instances as well as disjoint and abstract classes. In our approach we use an algebraic semantics [BH+99], but it can be easily shown that both semantics fit well together. In particular, sorts in order-sorted algebras satisfy the requirements imposed on sets modelling classes [BF98]. In general, algebraic semantics (see [Wi90]) is capable of specifying sets and, vice versa, algebraic models can be

expressed using set theory. The algebraic semantics requires an additional parameter env to formalize properties and operations. This parameter corresponds to the environment and can be interpreted as the above-mentioned valuation. A number of OCL semantics exist (cf. e.g. [RG98, CK01, Ba02, HKB04]). The closest to our approach is [HKB04].

9 Concluding remarks

In this paper we have defined an approach to redesign of UML class diagrams. It allows one to derive an interpretation function from a dependency relationship and to use such a function for an automatic transformation of OCL-constraints, provided that extendability conditions are satisfied. The concept of redesign is more general than the concept of refinement. We do not assume that system properties are merely added or refined, but we allow their change in an arbitrary way.

There are three crucial notions in our approach: compositionality, partiality and orthogonality. Compositionality concerns the formal model and assures that clients of a model element are not broken after redesign, i.e. that they may rely on properties they have used prior to the redesign. Partiality allows us to omit inessential parts of a model. In particular, we can neglect some parts of a model and treat a composed model element as atomic. Orthogonality and some monotonicity conditions allow us to extend a mapping to an interpretation function; such functions preserve different kinds of entailment relations. This is an important fact, since one might be interested in the preservation of logical relations between constraints; for example that an invariant implies a pre-condition or that a post-condition implies an invariant. Existence of an interpretation function saves the clerical work of redoing proofs after a transformation is performed.

In this paper we use compositional functions to define the notion of trace. A trace is an ordered set of model elements which are the antecedent or subsequent versions of those model elements. The ordering on model elements is defined by dependency relationships, or equivalently, by the generated compositional functions on the formal level.

The concept of order-sorted algebra has been extended to membership equational logic. We are going to investigate to which extent the results presented here can be extended to this logic. We are going to further study properties of the interpretation function in logical terms, in particular the relation to institutions. We plan also to implement a tool supporting class redesign, constraint transformation and tracing of model elements. Such a tool will be very helpful since a purely manual transformation of complex OCL-constraints is very laborious and error prone.

Graph rewriting proved to be a very powerful mean for studying different kinds of structure transformations. We are going to investigate how to generalize the results obtained in the case of graph rewriting.

References

[Ba02] Baar, T (2002) How to Ground Meta-Circular OCL Descriptions - a Set-Theoretic Approach. In Clark T, Evans A, Lano K (eds) ROOM 2002, The Fourth Workshop on Rigorous Object-Oriented Methods, King's College London

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

[BH+99] Bidoit, M, Hennicker, R, Tort, F, Wirsing, M (1999) Correct realizations of interface constraints with OCL. In France R, Rumpe B (eds) The UML - Beyond the Standard, Proc. 2nd Int. Conf, UML'99, LNCS 1723, Springer, Berlin, pp 399-415

[Bo+00] Bouhoula, A. Jouannaud, J.P, Meseguer, J (2000) Specification and Proof in Membership Equational Logic. TCS, 236(1-2), Elsevier

[BF98] J, M, Bruel, France, R (1998) Transforming UML models to formal specifications. In Mueller P, A, Bzivin J (eds) UML'98 Beyond the notation, LNCS 1618, Springer

[CK01] Cengarle, V, Knapp, A (2001) A Formal Semantics for OCL 1.4. In Gogolla M, Kobryn C (eds) UML 2001, LNCS 2185, pp 118-133

[CoFI98] CoFI Task Group Language Design (1998) CASL - The CoFI algebraic specification language. http://www.brics.dk/Projects/CoFI/

[DD01] Department of Defense (2001) A Defense System Software Development. DOD-STD-2167, http://jcs.mil/htdocs/teinfo/directives/soft/ds2167a.html.

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

[Eg03] Egyed, A (2003) Compositional and Relational Reasoning During Class Abstraction. In Stevens et al. (eds) Proceedings of the 6th International Conference on UML, San Francisco, USA, LNCS 2863, pp 121-137

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

[Eva+99] Evans, A, France, R, Lano, K. Rumpe, B (1999) Meta-modelling semantics of UML. In Kilov (ed) Behavioural Specifications for Businesses and Systems, Chapter 4, Kluwer Press

[FiMa91] Fiadeiro, J, Maibaum, T (1991) Describing, Structuring and Implementing Objects. In de Bakker F et al (eds) Foundations of Object Oriented languages. LNCS 489, Springer-Verlag

[Fin+94] Finkelstein, A, Gabbay, D, Hunter, A, Kramer, J, Nuseibeh, B (1994) Inconsistency Handling in Multiperspective Specifications, IEEE Transactions on Software Engineering, 20(8), pp 569 - 578

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

[Fra+02] France, R, Song, E, Kim, D-K, Ghosh S (2002) Using Roles for Pattern-Based Model Refactoring. Technical Report, Colorado State University

[GH+95] Gamma, E, Helm, R, Johnson, R, Vlissides, J (1995) Design Patterns. Addison-Wesley, Reading

[GR99] Gogolla, M, Richters, M (1999) Transformation Rules for UML Class Diagrams. In Bezivin J, Mueller P (eds) Proc. of UML'98, pp 92-106

[GM92] Goguen, J, Meseguer, J (1992) Order sorted algebra. Theoretical Computer

Science, 105(2), Elsevier, Amsterdam, pp 167-215

[Gr99] Große-Rhode, M. et al (1999) Refinements and modules for typed graph transformation systems. In Fiadeiro J (ed) Algebraic Development Techniques, ETAPS´98, LNCS 1589, Springer, Berlin, pp 137-151

[HKB04] Hennicker, R, Knapp, A, Baumeister H (2004) Semantics of OCL Operation Specifications. ENTCS 102, pp 111-132

[Hu+97] Hunter, A, Nuseibeh, B (1997) Analyzing Inconsistent Specifications. Proceedings of 3rd International Symposium on Requirements Engineering.

[Ki+05] Kirchner, C, Moreau, P, E, Reilles, A (2005) Formal validation of pattern matching code. Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming - PPDP'2005. Lisbon

[KP01] Knethen, A, Paech, B (2001) A Survey on Tracing. Approaches in Practice and Research. IESE-Report Nr. 095.01/E, Fraunhofer Institute

[Ja98] Jarke, M (1998) Requirements Tracing. Communications of the ACM, 41(12)

[Ko01] Kosiuczenko, P (2001) Formal Redesign of UML Class Diagrams. In Evans A, France R, Moreira A, Rumpe B (eds) Proc. of pUML Workshop, Toronto. GI-Edition, Lecture Notes in Informatics

[Ko05] Kosiuczenko, P (2005) Proof Transformation via Interpretation Functions. Technical Report Nr. 2004/27, University of Leicester, 16 pages
The new version of this report containing proofs mentioned in this paper can be found at: URL: http://www.cs.le.ac.uk/~pk82/Theory3.pdf

[La95] Lano, K (1995) Formal object oriented development. Springer, Berlin

[La96] Lano, K (1996) Formalizing Design Patterns. BCS-FACS Northern Formal Methods Workshop. Ilkley, UK, 23-24 September 1996, Springer

[Me98] Meyer, B (1998) Object-Oriented Software Construction. Prentice, Hall, N.J

[PR94] Paech, B, Rumpe, B (1994) A new concept of refinement used for behavior modeling with automata. In Proceedings of FME´94, LNCS 873, Springer, Berlin

[RG98] Richters, M, Gogolla, M (1998) On Formalizing the UML Object Constraint Language OCL. ER'98, Singapore, LNCS 1507, pp 449-464

[Ta99] Tarlecki, A (1999) Institution: An Abstract Framework for Formal Specifications. In Astesiano E, Kreowski H-J, Krieg-Brückner B (eds) Algebraic Foundations of System Specification, Springer

[Ta73] Taylor, W (1973) Characterizing Malcev conditions. Algebra Universalis, 3, Springer, Berlin, pp 351-397

[Te03] Terese et al (2003) Term rewriting systems. Cambridge Tracts in Theoretical Computer Science 55, Cambridge University Press

[UML05] OMG (2005) Unified Modeling Language Specification. Version 2.0, 05-07-04

[WK99] Warmer, J, Kleppe, A (1999) The Object Constraint Language. Addison-Wesley, Reading

[WN94] Watkins, R, Neal, M (1994) Why and How of Requirements Tracing. IEEE Software, July, pp 104-106

[Wi90] Wirsing, M (1990) Algebraic specification. In van Leeuwen J (ed) Handbook of Theoretical Computer Science. Elsevier, Amsterdam, pp 677-780

[ZMK05] Zawlocki, A, Marczynski, G, Kosiuczenko, P (2005) Property Preserving Redesign of Specifications. In Fiadeiro J, Rutten J (eds) CALCO'05, Swansea, LNCS 3269, pp 439-454

10 Appendix: formal background

In this section we define the basic algebraic notions used in our paper. We explain briefly the concept of order-sorted algebras. We define the notion of signature, order-sorted signature, term and sort. In the second part of this section, we explain how order-sorted algebras can be used to formalize inheritance. We explain also how to formalize OCL quantifiers over finite domains.

10.1 Algebraic background

An *algebraic signature* is a pair (S, F) where S is a set of sorts and F is a set of typed function symbols, e.g. $f : s_1,...,s_n \to s \in F$, for some $s_1,...,s_n, s \in S$. 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 [GM92]. We say that S has a *tree structure*, if one sort cannot be a subsort of two incomparable sorts; i.e. if $s \leq s_1$ and $s \leq s_2$, then $s_1 \leq s_2$ or $s_2 \leq s_1$, for every sorts s, s_1, s_2. This corresponds to the assumption that there is no multiple inheritance. A sort $s_l \in S$ is the largest sort in S if, and only if, for every sort $s \in S$, $s \leq s_l$.

For a function f, Dom(f) is the *domain* of f. We call a total function *mapping*. Let X be a set, let S be a set of sorts, and let $\tau : X \to S$ be a mapping. We call the elements of X variables. We say that a variable $x \in X$ is of type/sort s if, and only if, $\tau(x) = s$. When the typing function τ is clear, we write x : s instead of $\tau(x) = s$. We always assume that there are infinitely many variables of every sort.

The notion of term and term type is defined inductively: if $f : s_1,...,s_n \to s$, $t_1 : u_1,..., t_n : u_n$ and $u_i \leq s_i$, then $f(t_1,..., t_n) : s$; we write also $\tau(f(t_1,..., t_n)) = s$. The set of all terms with variables in X is denoted by $T(\Sigma, X, \tau)$. For a term t, var(t) is the set of variables contained in t. The expression $t(x_1,..., x_n)$ means that t contains at most the variables $x_1,..., x_n$; i.e. $var(t) \subseteq \{x_1,..., x_n\}$. We call a mapping $\sigma : X \to T(\Sigma, X, \tau)$ a *substitution*, if for every variable x : s, the type of $\sigma(x)$ is a subtype of s. We often write x^σ instead of $\sigma(x)$. The term $t[t_1/x_1,..., t_n/x_n]$ is obtained from t by applying the substitution $[t_1/x_1,..., t_n/x_n]$, which maps variable x_i to t_i, for i = 1,..., n, and leaves other variables of t unchanged. We call this operation *term composition*. We say that a substitution σ *preserves types*, if for every variable x, $\tau(x) = s$ implies that $\tau(\sigma(x)) = s$. We call a substitution *variable renaming*, if it substitutes variables for variables and preserves types.

An *order-sorted algebra* [GM92] over a signature Σ has the form $A = ((A_s)_{s \in S}, (f^A)_{f \in F})$; it consists of a family of non-empty carrier sets $(A_s)_{s \in S}$ such that $A_u \subseteq A_s$, for $u \leq s$, and a family of functions $(f^A)_{f:s1,...,sn \to s \in F}$ such that $f : s_1,...,s_n \to s^A: A_{s_1} \times ... \times A_{s_n} \to A_s$.

A *specification* is a pair Spec(Σ, Ax) consisting of an order-sorted signature Σ and a set of formulas Ax over the signature Σ.

In UML class diagrams associations may have multiplicity larger than one, in this case a function formalizing such association has values of the type Set(T). Every function of the

form $f : A_1 \times ... \times A_n \to B$ can be applied on the level of sets, i.e.

$f : Set(A_1) \times ... \times Set(A_n) \to Set(B)$ and f is defined as follows:

$f(X_1,..., X_n) =_{def} \{f(x_1, x_2,..., x_n) \mid x_1 \in X_1,..., x_n \in X_n\}$.

Similarly we can treat bags and sequences. However, for the sake of simplicity we assume that associations correspond to set valued functions and that a composition of two associations results in a set. We use boolean valued functions such as

_->exists(x | ...), _->forAll(x | ...), _->isEmpty(),...

for the equally named OCL operations on collections types (the definition is straightforward). It is worth noting that existential and general quantifiers over final domain can be specified algebraically as boolean valued functions. For example, ->exists(x | F(x)) can be specified by the following equations:

{}->exists(x | F(x)) = false,

$\{a_1\} \cup \{a_2,..., a_n\}$->exists(x | F(x)) = $F(a_1)$ or ($\{a_2,..., a_n\}$->exists(x | F(x))).

The OCL predefined operation `size()` corresponds to the set theoretic function |_|, which counts elements contained in a set. For a set A, |A| is equal to the number of elements contained in A. This function can be defined by induction as follows: $|\{\}| = 0$, and $|\{a_1,..., a_{n-1}\} \cup \{a_n\}| = |\{a_1,..., a_{n-1}\}| + 1$, if $a_n \notin \{a_1,..., a_{n-1}\}$. Note that the empty set plays the role of a constant constructor. Similarly, the operation $X \cup \{x\}$ plays the role of non-constant constructor. All sets can be obtained by applying those operations, e.g. $\{a_1,..., a_n\} = \{\} \cup \{a_1\} \cup ... \cup \{a_n\}$. The size of a bag and a sequence can be defined in a similar way.

10.2 Proof of the Extendability Theorem

In this subsection we prove the extendability theorem. Let the assumption of this theorem be satisfied. First, we extend φ to a relation ψ defined on the set $gen(A, X \cap X')$. Let ψ_0 be the smallest relation[6] such that for every variable renaming σ, if $u \varphi v$ holds, then $u^\sigma \psi_0 v^\sigma$ holds as well. We define ψ to be the smallest relation satisfying the following conditions:

- ψ contains ψ_0.
- ψ contains all pairs (x, x), such that $x \in X \cap X'$.
- If ψ relates the term t_i to the term t_i', for $i = 1,..., n$, $x_1,..., x_n \in X \cap X'$, $\psi_0(t_0) = t_0'$, the substitutions $[t_1/x_1,..., t_n/x_n]$ and $[t_1'/x_1,..., t_n'/x_n]$ are well defined, then ψ relates the term $t_0[t_1/x_1,..., t_n/x_n]$ to $t_0'[t_1'/x_1,..., t_n'/x_n]$.

One can easily prove by structural induction that if $u \psi v$, then $var(u) \subseteq var(v)$ (see condition (c) of the definition of orthogonal mappings). First, we prove that ψ is a partial function which maps variables on variables. Let us assume that it is not the case; i.e. that ψ

6. In set theory, relations and in particular functions are sets. The smallest relation means a relation included in all other relations.

relates a term t to two different terms: t ψ u and t ψ v. Let t be a term of minimal high, which has this property.

Composing any term with a nonvariable term results in a nonvariable term. Similarly, application of variable renaming to a nonvariable term results in a nonvariable term, and application of variable renaming to a variable results in a variable. Therefore a variable can be related by ψ only to itself. Consequently, t cannot be a variable.

From the definition of ψ and the fact that t cannot be a variable, it follows that there exist terms u_0, $v_0 \in A$, such that t can be presented in the form $u_0[u_1/x_1,..., u_m/x_m]$ and in the form $v_0[v_1/y_1,..., v_n/x_n]$, and that $u_0[u_1/x_1,..., u_m/x_m]$ ψ_0 u and $v_0[v_1/y_1,..., v_n/y_n]$ ψ_0 v. u and v have the form $\varphi(u_0)[\psi(u_1)/x_1,..., \psi(u_m)/x_m]$ and $\varphi(v_0)[\psi(v_1)/y_1,..., \psi(v_n)/y_n]$, respectively.

Since A is orthogonal and since u_0 and v_0 are unifiable, they must be identical. Consequently, we can assume that after suitable index permutation u_i is identical with v_i. Since u_0 and v_0 are mapped on the same term by φ and since t has two different images, there exist a number i such that u_i (or equivalently v_i) has two different images in respect to ψ. But this contradicts the assumption that t is a minimal term related to two different terms.

To prove the compositionality property we have to show that if ψ maps t_i to t_i' and $x_i \in X \cap X'$, for i = 1,..., n, the term $t_0[t_1/x_1,..., t_n/x_n]$ is well defined and $t_1,..., t_n \in gen(A, X \cap X')$, then the substitution $[t_1'/x_1,..., t_n'/x_n]$ is well defined also; i.e. $\tau'(\psi(t_i)) \leq' \tau'(x_i)$ for i = 1,..., n. Note that in general it is enough to prove that if ψ is defined on the term t, $x \in X \cap X'$, and $\tau(t) \leq \tau(x)$, then $\tau'(\psi(t)) \leq' \tau'(x)$. Note that $\tau(t) \leq \tau(x)$.

Let t be a variable y. Then $\tau(y) \leq \tau(x)$, $\psi(y) = y$ (see above), and $\tau'(\psi(y)) = \tau'(y) = \rho(\tau(y)) \leq' \rho(\tau(x)) = \tau'(x)$ (because of the conditions (b) and (e)).

Let us assume that the compositionality property is possessed by terms $v_1,..., v_m$. Let $x_1,..., x_m \in X \cap X'$ and let the term t have the form $v_0[v_1/x_1,..., v_m/x_m]$, for $v_0 \in A$. The type of a term is equal to the type of its top; i.e. $\tau(t) = \tau(v_0)$. Consequently $\tau(v_0) \leq \tau(x)$.

Let $\{x_1,..., x_m\} = var(v_0)$. Either $\varphi(v_0)$ is a variable or not. If it is not a variable, then $\tau'(\psi(t)) = \tau'(\varphi(v_0))$ and

$$\tau'(\psi(t)) = \tau'(\varphi(v_0)[\psi(v_1/x_1),..., \psi(v_m/x_m)]) = \tau'(\varphi(v_0)) = \rho(\tau(v_0)) \leq' \rho(\tau(x)) = \tau'(x).$$

If $\varphi(v_0)$ is a variable then $\varphi(v_0)$ has the form x_j, since t is well defined and $var(\varphi(v_0)) \subseteq var(v_0)$ (see the condition (c)). Consequently $\tau(v_j) \leq \tau(x_j)$. Because of the inductive assumption $\tau'(\psi(v_j)) \leq \tau'(x_j)$ holds. Consequently because of the inductive assumption and conditions (b), (d) and (e) the following holds:

$$\tau'(\psi(t)) = \tau'(\varphi(v_0)[\psi(v_1/x_1),..., \psi(v_m/x_m)]) = \tau'(\psi(v_j)) \leq' \tau'(x_j) =$$
$$\tau'(\varphi(v_0)) = \rho(\tau(v_0)) \leq \rho(\tau(x)) = \tau'(x)$$

The fact that the domain of ψ is equal to gen(A, $X \cap X'$) follows from the fact that ψ is compositional, since the domain of ψ is the smallest set containing A as well as $X \cap X'$, and closed on term composition.

The proof of uniqueness follows by structural induction. Let θ be another compositional function extending φ. θ and ψ must coincide on variables from $X \cap X'$ and on terms from A. Let us assume that those functions coincide on terms r, t_1,..., t_n. In turn, they must coincide on the term $r[t_1/x_1,..., t_n/x_n]$ because of compositionality. ♦

10.3 OCL and inheritance

In this subsection we briefly explain, how order-sorted algebras can be used to formalize inheritance in object-oriented systems. We also explain how to formalize sets and quantifiers over finite domains.

The requirement that equal terms are of the same sort corresponds to the Java 1.4 assumption, that if a method in a subclass overwrites a method in a super-class, then it cannot change the return type of the overwritten method. In Java methods, with different explicit parameters, are considered to be different, event if their implicit parameters have the same type (it is the so-called method overloading). In this framework, method overloading can be indirectly dealt with using methods with different names.

Order-sorted algebras do not allow us to reason about method overriding; they allow us only to refine them. For example, let `m` be a method which has an implicit parameter of type `A`, an explicit parameter of type `C` and which returns objects of class `D`. Let `B` be a subclass of `A`, then `m` can be overwritten for this subclass. The method acting on class `A` can be modelled by a function m : Env × A × C → Int. Its restriction to the subsort B of sort A, i.e. m : Env × B × C → Int, can possess some additional properties. It can satisfy constraints that are not satisfied by the general function. For example, let us consider the following two formulas:

$$(e : Env, a : A).m(c : C) < 10, \qquad (e : Env, b : B).m(c : C) > 0$$

From these formulas, it follows that the value returned by m is smaller than 10. The restricted function returns positive values, if the actual implicit parameter is of sort B, but the general function may return negative values as well.