Refactoring Information Systems

– A Formal Framework –

M. Löwe, H. König, M. Peters, and Ch. Schulz
FHDW Hanover, Freundallee 15
D-30173 Hanover, Germany

ABSTRACT

We introduce a formal framework for the refactorization of complete information systems, i. e. the data model *and* the data itself. Within this framework model transformations are uniquely extended to the data level and result in data migrations that protects the information contained in the data. The framework is described using general and abstract notions of category theory. Two concrete instances of this framework show the applicability of the abstract concept to concrete object models. In the first instance, we only handle addition, renaming and removal of model objects. The second instance can also handle folding and unfolding within object compositions. Finally, we discuss how an instance of the framework should look like that is able to handle inheritance structures as well.

Keywords: Refactoring, Migration, Graph transformation, Pullback complement

1. INTRODUCTION

The only constant thing is change. This is especially true for the information and communication business. Currently, information systems in many companies are subject to change. This is mainly due to the technological progress connected to the Internet which enables completely new sorts of electronic business. Thus, we see big efforts to re-engineer the technical basis on the one hand and to improve the business processes and information models on the other hand [1].

This development has been reflected in the research and development community in the last years. Agile and Extreme Programming Techniques [2] [3] [4] aim at supporting the ongoing reengineering processes by providing refactorization methods, techniques, patterns [5][6] and tools [7]. These tools enable consistent global changes of a whole software system, for example to introduce some design patterns which are necessary for the system to take the next evolution step. This puts the flexibility into the development process that is needed to keep a system up-to-date (without any over-specifications at the beginning of the development) and to realize changing requirements quickly.

For the time being, the agile techniques are mainly restricted to the improvement and change of software systems within the development phase. There are only few attempts to apply the methods to running information systems and to extend the agile perspective to the maintenance and production phase of a software system's life cycle [8] [9]. The main obstacle for agile techniques here is existing data. The bigger the database, the larger the problems. If a model of a running information system is changed, we are faced with one central question: "What shall we do with the data conforming to the old model?" Up to now, we hear two major answers:

1. Leave the data as it is and map the new model to the old one using for example some object-relational-mapping tools [10].[1]
2. Migrate the data from the old model to the new one by crafting corresponding migration scripts and performing the (long-running) data migration at night or on the weekend.

Both solutions possess big disadvantages. The first one leads to complex mappings if applied several times. This complexity is very likely to produce performance problems and reduce the development speed of the engineering team in the long run.[2] The second solution requires long production breaks and consumes a lot of development and test time for software (migration scripts) that is thrown away after success.

We propose another approach, namely the generation of the necessary data migration directly from model refactorizations, compare also [11]. The actual migration can be performed for the whole database at once as in the second approach above. We save the development costs in this scenario. If the migration can be performed on demand[3] for single records – only if a record is touched which has not been migrated yet – we are also able to avoid production breaks. The research and development program necessary to elaborate the theoretical basis and a corresponding tool support for this approach is ambitious. We are just starting it within a project supported by the FHDW Hanover.

One central issue is the *correctness* of the induced migrations. We can only benefit from this approach if we can trust in the produced migrations without any further tests. Therefore, we present a theoretical framework in this paper, which

1. is able to represent models and instances in a uniform meta-model,
2. comes equipped with a suitable notion of semantics-preserving model refactorization,
3. provides refactorization-induced correct transformations of the instances (migrations), and
4. proves its applicability by satisfying necessary and natural properties for refactorizations and migrations.

The general and abstract framework, which is presented in section 2, is built on category theory [12]. In this theory, we not only have a very general notion of structured *object*. By the notion of *morphism*, we also get a natural way of representing (1) typings of instance objects in model objects as well as (2) model changes and instance migrations. Even and just on this

1 An older and worse version of this approach is: Leave the data-model as it is and redefine the meaning of the data within the model, for example by using comma-separated multi-value fields in a single string column.

2 The longer this approach is applied, the bigger the problems to switch to the second one.

3 Comparable to load-on-demand-techniques for instantiating objects from relational database rows.

very abstract level, we can determine the properties *any* reasonable refactorization framework shall possess. We demonstrate this by evaluating the properties for refactorizations to possess a consistent notion of sequential composition.

Section 3 and 4 provide instances of the framework, where objects are graphs and morphisms are some sort of graph-structure-preserving mappings as being outlined in the theory of algebraic graph transformation [13].[4]

Section 3 provides an instance of the framework that can handle addition, renaming, and deletion of model objects only. The formal description is straightforward.

Section 4 introduces a model that can handle splitting and gluing of model objects within composition structures. Here it is possible to redirect associations as long the source and the target remain in the same composition structure. The complex construction in this section demonstrate the increase of complexity if we want to extend refactorizations from models to whole information systems.

Finally, we sketch in section 5 how inheritance structures (like for example in the UML-class-models [14]) and their evolution can be handled within this framework.

All proofs in this paper are only sketched since space is limited. For detailed proofs see [15].

2. CATEGORICAL FRAMEWORK

Category theory is appropriate for our purposes for two reasons. First we can express the typing of some data D in a model M by a morphism $t: D \to M$. The morphism assigns to each data element its type. Second we need to express refactorizations between models and migrations between typed data. Since we are not only interested in the model states before and after refactorization but in the refactorization process itself, it is also a good choice to represent a model refactorization from model M to model N by morphisms: $M \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} N$. The pair ($l$, r) represents an arbitrary relation between M and N and can model:

1. Deletion of model objects, i. e. l is not surjective,
2. Addition of model objects, i. e. r is not surjective,
3. Renaming of model objects, i. e. l and r are bijective but not identities,
4. Splitting of model objects, i. e. l is not injective, and
5. Gluing of model objects, i. e. r is not injective.

Since we do not require the same properties for typings and refactorizations in general, we distinguish a special class of morphisms for each purpose.

Given a typed database $t: D \to M$ and a model refactorization $M \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} N$, we want to canonically construct the induced migration to some typed database $u: E \to N$. As a first step, we can use the pullback construction of t and l, which shall result in a typed database $v: F \to K$. In the second step, we need to construct a pullback complement of v and r. Unfortunately, such a pullback complement is not guaranteed to exist nor need be unique if it exists. Thus we have to require the unique existence of some special pullback complements for our framework.

Definition 1 (Migration Framework). A *migration framework* is a category C together with two subcategories T and R which have the same objects as C. The morphisms in T are called *typings* and the morphisms in R are called *modifications*. The system (C, T, R) is subject to the following requirements:

(A1) C and R have pullbacks and pullbacks in R are pullbacks in C as well.

(A2) Pullbacks in C preserve epimorphisms.

(A3) If $(l': F \to D,\, v: F \to K)$ is the pullback of the pair of morphisms $t: D \to M \in T$ and $l: K \to M \in R$, then $l' \in R$ and $v \in T$.

(A4) Each morphism pair $(v: F \to K \in T,\, r: K \to N \in R)$ has a unique (up to isomorphism) pullback complement $(r': F \to E,\, u: E \to N)$[5], such that r' is epimorphism and $r' \in R$ and $u \in T$. Such a pullback complement is called *initial pullback complement*.

Definition 2 (Migration). Let (C, T, R) be a migration framework. A *model refactorization* is a pair of R-morphisms $(l: K \to M,\, r: K \to N)$. The *application* of a model refactorization $R = (l: K \to M,\, r: K \to N)$ to a typing $t: D \to M \in T$ is defined by the *migration diagram* to the right, where (1) is pullback of l and t, and (2) is initial pullback complement of r and v. The result is u.

$$\begin{tikzcd} M & K \arrow[l, "l"'] \arrow[r, "r"] & N \\ D \arrow[u, "t"] \arrow[ur, phantom, "(1)"] & F \arrow[l, "l'"] \arrow[r, "r'"'] \arrow[u, "v"] \arrow[ur, phantom, "(2)"] & E \arrow[u, "u"'] \end{tikzcd}$$

Note that v is typing due to axiom A3 of the migration framework. Therefore, axiom A4 guarantees the existence of the initial pullback complement diagram (2).

Proposition 3 (Uniqueness of Migration). The system u that is the result of migrating the system t by refactorization R is uniquely determined up to isomorphism. Therefore we can write $R(t) = u$.

Proof. Direct consequence of the axioms of a migration framework and the construction of a migration.

The axioms of a migration framework are strong enough to show that the result of applying the sequential composition of two model refactorizations coincides with the system which we obtain if we apply the two refactorizations sequentially. The rest of this section is dedicated to this result.

Definition 4 (Sequential Composition). The *sequential composition* $R_2 \circ R_1 = (l_1 \circ p_1: J \to M,\, r_2 \circ p_2: J \to P)$ of two refactorizations $R_1 = (l_1: K \to M,\, r_1: K \to N)$ and $R_2 = (l_2: H \to N,\, r_2: H \to P)$ is defined by the pullback object $(p_1: J \to K,\, p_2: J \to H)$ of r_1 and l_2, see below:

$$\begin{tikzcd} & & J \arrow[dl, "p_1"'] \arrow[dr, "p_2"] \arrow[dd, phantom, "(PB)"] & & \\ M & K \arrow[l, "l_1"'] \arrow[dr, "r_1"'] & & H \arrow[dl, "l_2"] \arrow[r, "r_2"] & P \\ & & N & & \end{tikzcd}$$

Fig. 1: Sequential composition

Note that the sequential composition is well-defined due to axiom A1 of migration frameworks.

4 Graphs or some generalization of graphs can be and have been used to generalize object-oriented and entity-relationship models.

5 $A \stackrel{b'}{\rightarrow} D \stackrel{a'}{\rightarrow} C$ is pullback complement of $A \stackrel{a}{\rightarrow} B \stackrel{b}{\rightarrow} C$ if (a,b') is pullback of (b,a').

Theorem 5 (Sequential Composition). If the sequential composition $R_2 \circ R_1$ of two refactorizations is defined, we have $R_2 \circ R_1(t) = R_2(R_1(t))$.

Proof Sketch. Consider the following diagram, which depicts $R_2 \circ R_1(t)$ as well as $R_2(R_1(t))$:

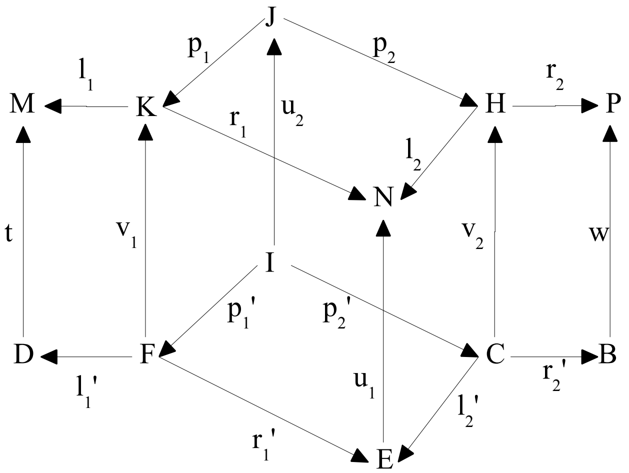

Fig. 2: Migration sequence

Let us first consider $R_2(R_1(t))$. This migration sequence is given by the four squares (MKFD), (KNEF), (NHCE), and (HPBC). The morphisms (p_1,p_2) are given as pullback morphisms by the construction of $R_2 \circ R_1$. We construct (p_1',p_2') as pullback of (r_1',l_2'). The morphism u_2 is the universal completion of the diagram into the pullback object J. Now, the squares (KJIF) and (JHCI) become pullbacks as well. Since r_1' is epi, p_2' is epi as well by axiom A2. Thus, (JHCI) is initial pullback complement of u_2 and p_2. By uniqueness (Def. 1, A4) and the fact that epimorphisms compose, (w, r_2' o p_2') is initial pullback complement of (u_2, r_2 o p_2). Since pullbacks compose, (u_2, l_1' o p_1') is pullback of (t, l_1 o p_1). This together shows that $R_2 \circ R_1$ migrates *t* to *w* as well.

Theorem 5 and its consequences are necessary for any reasonable migration framework:

1. Several refactorization steps can be composed. Instead of performing many simple migration steps sequentially, the result can be obtained by a single more complex migration.
2. Vice versa, complex migrations can be decomposed into more elementary steps. (This might be a topic for further research, finding atomic migrations.)
3. This can be the basis for some sort of migration scripts. A sequence of migration steps performed to one information system can be “recorded” in a composition. After that it can be “replayed” in a single step to another system.
4. Sequential composition can be improved if we omit inverse atomic steps in two consecutive refactorizations.
5. Sequential composition does not result in effects that can not be observed using the component steps.

There are several other results which can be obtained on this very abstract level and hold for any instance of the framework.[6] In the rest of the paper, we demonstrate that there are instances that are interesting from the theoretical as well as from the practical point of view.

3. ADDITION, RENAMING, AND REMOVAL

In this section, we provide the first instance of the migration framework in the sense of definition 1. The basis, also for the instance in the next section, is the following category of simple graphs.

Definition 6 (Graph). The category **G** of *graphs* is the algebraic category w. r. t. the following signature:

Sorts: Object
Opns: source, target: Object → Object .

This is a simple form of graphs where we do not distinguish between nodes and edges. In such a graph, nodes can be characterized as objects *n* such that source(*n*) = *n* = target(*n*). Graphs and graph morphisms of this type provide more flexibility in the refactorization/migration context we are considering here, for example: if two nodes *x* and *y* are mapped to the same node *z*, it is possible that a morphism maps an edge *e* with source(*e*) = *x* and target(*e*) = *y* to *z* as well.

Definition 7 (Graph as a migration framework ARR). The migration framework **ARR** is given by the following system of categories **ARR** = (**G**,**G**,$\mathbf{G}^{\mathrm{I}}$), where $\mathbf{G}^{\mathrm{I}}$ is **G** restricted to monomorphisms only.

Note that all axioms of a migration framework are satisfied. **G**, like all algebraic categories, has all limits. Pullbacks preserve monomorphisms. Thus A1 and A3 are satisfied. There is a forgetful functor V: **G** → **Set** and epimorphisms coincide with retractions in Set. Since V preserves limits and pullbacks preserve retractions, pullbacks in **G** preserve epimorphisms (A2). To show axiom A4, let a:A → B and b:B → C be given and b be injective. We construct the initial pullback complement as (A, id_A, b o a). That this results in a pullback diagram is easily checked. Since b is mono, in any initial pullback complement (X, c:A → X, d:X → C), c must be mono. The morphism c being mono and epi gives an isomorphism. Hence, X is isomorphic to A, as desired.

The instance ARR allows to delete model objects (left-hand side of the refactorization is not epi). The corresponding migration deletes all instances of the removed model object. Model objects can also be added (right-hand side of the refactorization is not epi). The corresponding migration does not generate any instances at all and leaves the instance sets of the added model objects empty. The possibility of renaming is always given, since neither the left-hand nor the right-hand side of a refactorization is required to be an inclusion.

The nice properties of **ARR** can be re-obtained on the categorical level, if we allow arbitrary morphisms for the left-hand sides of refactorizations, but stick to injective morphisms on the right-hand side. Then we get the possibility of copying objects during the migration. But we do not have any inverse operation, that glues together several copies of the same object. This seems not reasonable from the application point of view, especially because we do not possess any operation that compensates or undoes wrong copy processes.

But non-injective morphisms as right-hand sides of refactorizations must be handled with care. In the general case, we do not have initial pullback complements as the following counter-example demonstrates. The right-hand side being an

6 Another example is vertical composition: If we stack migrations vertically, we obtain a simple migration as well, compare [15].

epimorphism is not sufficient as some authors erroneously argue [17].[7]

Example 8 (Ambiguous Pullback Complements). In the situation depicted in Fig. 3, epimorphism f and morphism g do not possess an initial pullback complement in **G**, since (g,f_1^*) is pullback of (f, g_1^*) and (g,f_2^*) is pullback of (f,g_2^*) but D_1 and D_2 are not isomorphic:

There seems to be no chance to avoid this type of ambiguity, if we do not put additional requirements on the “vertical” morphisms g, g_1^* and g_2^*, the typings in our case. These properties shall single out a unique choice for the pullback complement extension of g. This approach has been prepared by axiom A4 of definition 1 and is elaborated in the following section.

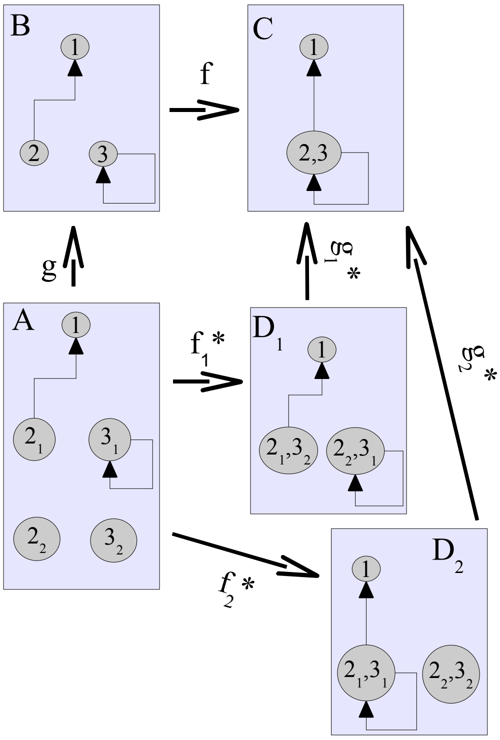

Fig. 3: Ambiguous pullback complements

4. COPYING AND MERGING

Example 8 demonstrates that arbitray gluing in a refactor-ization's right-hand side introduces ambiguity for the in-stantiation of the gluing in the induced migration. This type of ambiguity cannot be accepted in a reasonable migration framework. There seems to be no way of controlling this ambiguity, if we put requirements on the modifications only. If we use restrictions for the typings there might be a chance to single out a unique gluing on the migration level.

This idea can be made more concrete using example 8 again: If we knew, that gluing is only permitted within *components* and the object 2_1 together with object 3_1 resp. 2_2 together with 3_2 form components in graph A, we can guess that (f_2^*,g_2^*) is the right pullback complement. On the other hand, if the components in graph A are given by $\{2_1,3_2\}$ and $\{2_2,3_1\}$, the right choice is (f_1^*,g_1^*). These informal ideas of “components”, “intra-component-gluing”, and typings that “respect components” are made precise in the rest of this section.

7 See also [16] for a discussion of pullback complements of graphs.

Definition 9 (Component Graph). A *component graph* (G,P,i) consists of a graph G, a system of part sets P, and a family of embeddings $i=(i_p : p \to G)_{p\in P}$, such that the following two requirements are met:

1. (G_{Object}, i) is the sum of P, written $(G,i)=\sum P$.
2. If $x \sim y$, source(x) ~ source(y) and target(x) ~ target(y), where $x \sim y$ if x and y are in the same part, which means that $x = i_p(x')$ and $y = i_p(y')$ for some $p \in P$.

A morphism $(f_G, f_P):(G,P,i)\to(H,Q,j)$ is a morphism $f_G : G\to H$ on graphs together with a family $f_P=(f_p : p\to q^p \in Q)_{p\in P}$ of mappings, such that

3. $(f_G \circ i_p = j_{q^p} \circ f_p)_{p\in P}$.

The category **CG** consists of all component graphs and their morphisms. ■

Remarks.

For any component graph (G,P,i), P is a decomposition of the carrier set G_{object} (up to isomorphism).

Each component graph morphism maps components in the domain to components in the co-domain.

The abstract concept of components, presented here, is often realized by some special sort of associations: objects p and q belong to the same component if there is a path of symmetric one-to-one associations from p to q.

Definition 10 (CG as a migration framework CM). The category **CG** of component graphs can be turned into a migration framework **CM** = (**CG**, **CG**^{T}, **CG**^{R}), if

1. for all *typings* (f_G, f_P) in **CG**^{T}, f_P is a family of isomorphisms and
2. for all *modifications* $(f_G, f_P):(G,P,i)\to(H,Q,j)$ in **CG**^{R}, f_P is a family of epimorphisms and (i_p, f_p) is pullback of (f_G, j_{q^p}) for each part p in G.

Remarks. Requirement (1) for typings, on the one hand, guarantees that any component in a model is completely instantiated in any instance, and that two objects that map to the same model object are identical or in different components. Requirement (2) for modifications, on the other hand, realizes the idea that gluing can only occur within components, since it states that two model objects that are mapped to the same object by a modification must be in the same component.

The axioms A1, A2, and A3 for migration frameworks are satisfied by **CM** due to corresponding properties of the underlying categories of graphs and sets. To show axiom A4, we provide an explicit construction of the initial pullback complement in the required situation.

Construction 11 (Pullback complements in CM). Let a typing $(t_G, t_P):(F,P,i)\to(G,Q,j)$ and a modification $(r_G, r_P):(G,Q,j)\to(H,R,k)$ be given. Then we construct the pullback complement (I,S,l), the modification $(r_G', r_P'):(F,P,i)\to(I,S,l)$, and the typing $(t_G', t_P'):(I,S,l)\to(H,R,k)$ for each part p in P as follows:

1. $p_S = r(t(p))$ is a copy of the part $r(t(p))\in R$,
2. $r'_p : p\to p_S ::= r\circ t$, and

3. $t'_{p_S}: p_S \rightarrow r(t(p)) ::= id_{r(t(p))}$.

Now let

4. the carrier $I = \sum_{p \in P} p_S$,

5. $r' : F \rightarrow I = \sum_{p \in P} r'_p$, and

6. $t' : I \rightarrow H = \sum_{p \in P} t'_{p_S}$.

The operation structure on I can be defined by:

7. $source^I(x) = r'(source^F(y))$, where $x = r'(y)$ and

8. $target^I(x) = r'(target^F(y))$, where $x = r'(y)$.

It is well-defined due to definition 9 (2). By this definiton r' becomes a graph morphism. By construction, we have $t' \circ r' = r \circ t$. Thus, t' is a graph morphism and a typing since it is the sum of identities. The morphism r' is epi on each part and as a graph morphism by construction, and – again by construction – the inverse image of each part in I is a part in F. Therefore, r' is a modification.

Proposition 12 (Pullback complements in CM). The construction 11 provides essentially unique initial pullback complements.

Proof Sketch. The constructed square commutes. That it is a pullback, can be easily checked using the part structures on each graph and the fact that t' is a sum of part identities and the kernel of r is contained in the decomposition Q on G. Using this fact again and that t is a sum of isomorphisms, we obtain that the kernel of $\overline{r}$ in every pullback complement $(\overline{r} : F \rightarrow J, \overline{t} : J \rightarrow H)$, in which $\overline{t}$ is a typing, coincides with the kernel of r'. Thus, the constructed square is the only (up to isomorphism) initial pullback complement.

Although **CM** allows copying[8] and gluing of objects only within the same component, it provides some interesting features for our purposes of information system refactorization, as the following example shows.

Example 13 (Redirection of associations). Consider the model refactorization in the picture below:

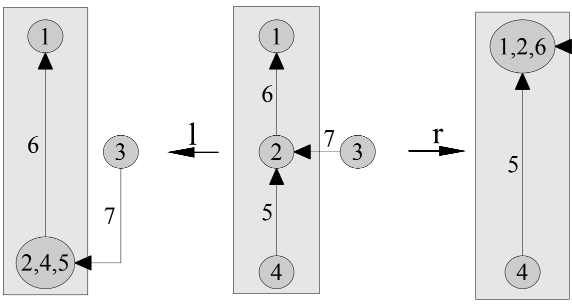

Fig. 4: Redirection of associations

All three graphs have 3 components; the non-trivial component in each graph (that has more than one element) is highlighted. Using this refactorization in a migration redirects all associations of type "7" from the source of "6" to the target of "6". It uses an intermediate vertex "2", that is introduced by the left-hand side "l" as an unfolding and removed again by the right-hand side "r" by a corresponding folding. This example shows, that we are able to redirect association sources and targets in **CM** as long as we stay in the same component.

5. CONCLUSIONS: HANDLING INHERITANCE

The formal framework we have developed in this paper is able to handle a broad spectrum of model refactorizations: Addition, deletion, and renaming of arbitrary model objects as well as foldings and unfoldings of model objects which belong to the same component. We have shown that all refactorizations consisting of these types of "elementary actions" induce a unique and consistent migration of existing instances from the old to the new model.

In order to be applicable to real-world object-oriented systems, however, it should be able to handle inheritance structures as well. Inheritance is some sort of static composition between objects: an object of class c can be considered to be composed of a set of (sub-)objects, namely one object for each direct or indirect ancestor class c' of c.[9] All these objects are created at the same time the most special object is created. And they are also destroyed at the same time. Hence, we can model them as explicit parts in a component graph on the instance level in our framework .

But these components are not components in the sense of typings in **CM**. It is not the case, that the *complete* inheritance tree of classes needs to be instantiated, if one class is. In general, each object is a proper subpart of the complete inheritance tree of its class, only. Our approach is not able to handle those incomplete parts. Many examples show that pullback complements (axiom 4) are not guaranteed to exist in this situation.

We, therefore, have to use a trick to handle inheritance. We always instantiate complete inheritance graphs, when an object is created and keep the information about the most special *real* object in the resulting part (of *real* and *extra* objects). Then we distinguish two views on the system: (1) the *refactorization perspective* and (2) the *operational perspective*. In the first perspective, all objects are visible and our framework is applicable. The second perspective blends out all extra objects in order to keep the system's state consistent from the operational point of view.[10]

With these additional arrangements, the formal model presented in this paper seems fit for practical applications.

REFERENCES

[1] Havey, M., Essential Business Process Modeling, 2005.
[2] Martin, R. C., Agile Software Development, Principles, Patterns, and Practices , 2002.
[3] Beck, K., Extreme Programming Explained, 2000.
[4] Beck, K., Test-driven Development by Example, 2002.
[5] Fowler, M., Refactoring: Improving the Design of Existing Code , 1999.
[6] Kerievsky, J. , Refactoring to Patterns, 2004.
[7] D'Anjou, J et al, The Java Developer's Guide to Eclipse, 2005.
[8] Ambler, S. W., Agile Database Techniques, 2003.
[9] Ambler, S. W., Refactoring Databases : Evolutionary Database Design , 2006.

8 Note that all copies of an object are put into the same component as the original object, since also the left-hand sides of refactorizations must be modifications. Thus, every copy process can be made undone by an inverse modification.

9 This interpretation is often used when object models are mapped to relational database systems using the "one table per class"-strategy. This strategy provides one relational table for each class and maps each inheritance association to a foreign key relation from the special to the general class.

10 Note that the model is stable under the operational perspective!

[10] Bauer, Ch., King, G. , Hibernate in Action , 2004.
[11] Löwe, M., Evolution Patterns – A Graphical Framework for Software Redesign, ISAS'99, 1999
[12] Adamek, J., Herrlich, H., Strecker G. E., Abstract and Concrete Categories - the Joy of Cats, 2006, http://katmat.math.uni-bremen.de/acc/acc.pdf
[13] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G., Fundamentals of Algebraic Graph Transformation, 2006.
[14] Larman, C., Applying UML and Patterns, 2005.
[15] König, H., Löwe, M., Peters, M., Schulz, Ch., A Formal Framework for Information System Refactorization, FHDW Hanover, Freundallee 15, D-30173 Hanover, 2006 (in German).
[16] Bauderon, M., Jacquet, H., Pullback as a generic graph rewriting mechanism, Applied Categorical Structures Vol.9(1), 2001.
[17] Meisen, J., Pullbacks in Regular Categories, Canad. Math. Bull. Vol.16(2), 1973.