Electronic Communications of the EASST
Volume 3 (2006)

Proceedings of the
Third Workshop on Software Evolution
through Transformations:
Embracing the Chance
(SeTra 2006)

Refactoring Information Systems

Michael Löwe, Harald König, Michael Peters, and Christoph Schulz

17 Pages

Guest Editors: Jean-Marie Favre, Reiko Heckel, Tom Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst ISSN 1863-2122

Refactoring Information Systems
- Handling Partial Compositions -

Michael Löwe, Harald König, Michael Peters, and Christoph Schulz

FHDW Hannover, Freundallee 15
D-30173 Hannover, Germany

Abstract: We present our formal framework for the refactoring of complete information systems, i.e., the data model and the data itself. It is described using general and abstract notions of category theory and can handle addition, renaming and removal of model objects as well as folding and unfolding within complete and partial object compositions.

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 refactoring 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, agile techniques in database engineering were often restricted to the improvement and change of model artifacts. The main obstacle for agile techniques here is existing data. Attempts to describe semantics-preserving schema transformations that also migrate data can be found in [8] [9]. A transformational approach that considers the instance level is discussed in [10].

If a model of a productive 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 [11].[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 refactoring, compare also [12]. 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 model refactoring,
3. provides refactoring-induced correct transformations of the instances (migrations), and
4. proves its applicability by satisfying necessary and natural properties for refactorings and migrations, i.e., that refactoring can be composed in a natural way.

The framework is built on category theory [13] and algebraic graph transformation [14]. 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 (refactorings) and instance migrations.

Section 2 presents our current framework built on a double-pullback construction, which can handle addition, renaming, and removal of model objects as well as folding and unfolding within *complete* object compositions [15]. This framework is not able to handle inheritance structures directly. Section 3 provides a slight generalization: We do not longer require that the right-hand side of a migration is a pullback. Instead we re-use the explicit construction of the pullback complements in more general situations. It turns out that this construction enjoys some categorical properties that guarantee uniqueness up to isomorphism. Section 4 shows that the usual sequential composition of refactorings extends to migrations in the generalized framework as well. We sketch in section 5, how the results in this paper can be reformulated on a purely categorical level. We explicitly point out the similarities to the approach of Ehrig et al. using adhesive categories [14]. Section 6 provides a conclusion and contains hints for future research activities.

1 An older and worse version of this approach is: Leave the data-model and the corresponding data 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.

2. Migration Framework

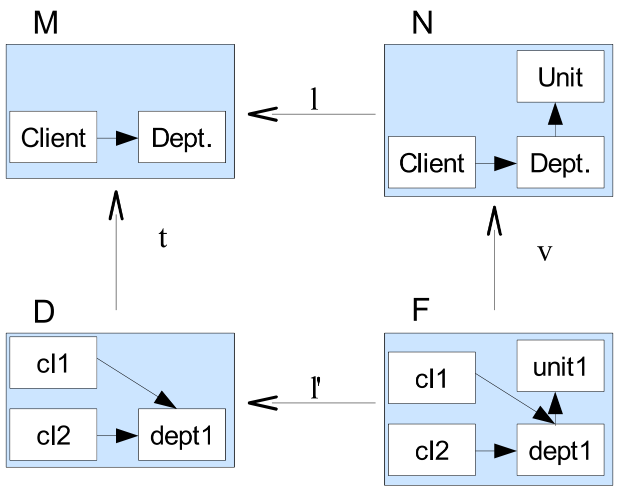

Fig. 1: Extracting a superclass

For a motivation of the following theoretical aspects, consider the situation of a class *Department*. A possible refactoring would be to extract an abstract superclass *Unit* in order to be prepared for additional specializations [5]. If we interpret generalization as object composition on the instance level, an automated migration must add a *Unit*-Object to each *Department*-Object in a 1:1 fashion[3]. After the migration client objects no longer use a single *Department*-Object but a new object which contains the *Unit*-information as an aggregated object, see Fig. 1. Since *Unit* is an abstract class, we can model this refactoring by unfolding *Department* to two classes. This can be done by a morphism *l* that maps the new model *N* to the old model *M* assigning the two classes *Department* and *Unit* in *N* to *Department* in M. Having data *D* which is typed by the morphism $t: D \rightarrow M$ we obviously can generate the migrated data by calculating the pullback object F of *t* and *l*[4].

Another possible refactoring is the addition of a new class [5]. This can be achieved with a (non-surjective) map *r* from the old to the new model, see Fig. 2. Here the question arises which categorical construct generates a reasonable migration. Moreover, different data structures are possible after the migration: one possibility would be to create no *B*-object, another to create a default-value or prototype object for *B*. Both solutions lead to pullback diagrams, if objects *a1* and *a2* are preserved.

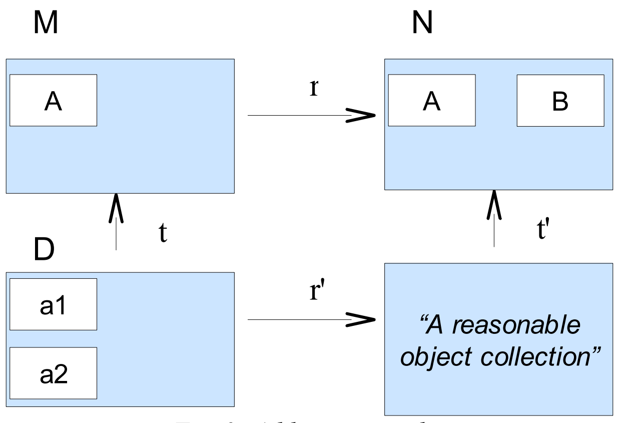

Fig. 2: Adding a new class

Category theory can be applied in the following way. We can express the typing of some data *D* in a model *M* by a morphism $t: D \rightarrow M$. And we need to express refactorings between models and migrations between typed data. We will have to use the two variations *l* and *r* discussed above. But we are not only interested in the model states before and after

3 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.

4 Later, we discuss in which category the construction is carried out.

refactoring but in the refactoring process itself. Hence, it is a good choice to represent *one* model refactoring from model M to N by a combination of the two variants, i.e., by *a pair* of morphisms: $M \xleftarrow{l} K \xrightarrow{r} N$. 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 or unfolding of model objects, i.e., l is not injective, and
5. Gluing or folding of model objects, i.e., r is not injective.

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

Given a typed database $t: D \rightarrow M$ and a model refactoring $M \xleftarrow{l} K \xrightarrow{r} N$, we want to canonically construct the induced migration to some typed database $u: E \rightarrow N$. As a first step, we can use the pullback construction of t and l, which shall result in a typed database $v: F \rightarrow K$. For reasons of symmetry, we need to construct a pullback complement of v and r in the second step. Unfortunately, such a pullback complement is not guaranteed to exist nor need be unique if it exists (see Fig. 2).

Even worse, there is no simple property for r that guarantees existence and uniqueness of the pullback complement. Some authors argue that r being epimorphism is sufficient, compare [16] or [17]. This is wrong as the following examples demonstrate.

Example 1 (Ambiguous Pullback Complements). Consider the situation depicted in Fig. 3 in the usual category of graphs and graph morphisms. The epimorphism f and the morphism g do not possess a unique (up to isomorphism) pullback complement, 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. □

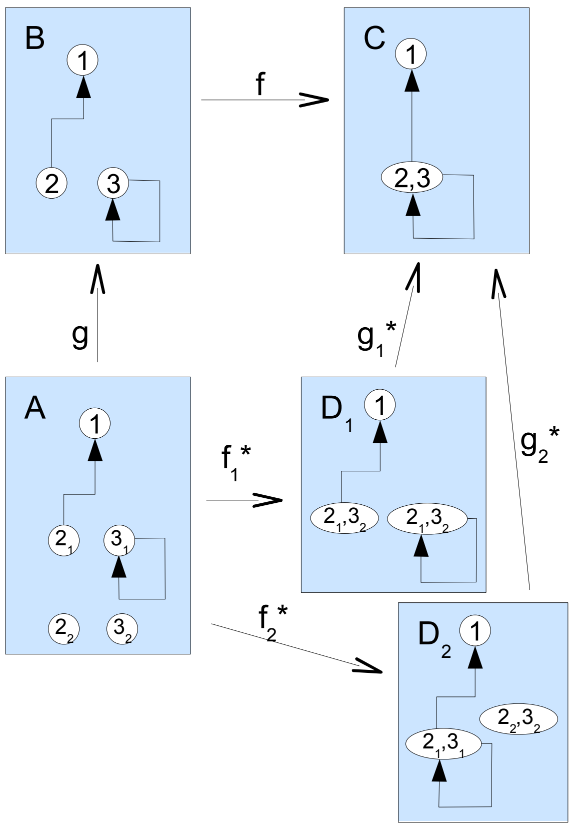

Fig. 3: Ambiguous Pullback Complement

In the category of sets and mappings, however, pullback complements seem to be uniquely determined. This is not (really) true, as is demonstrated by the following example.

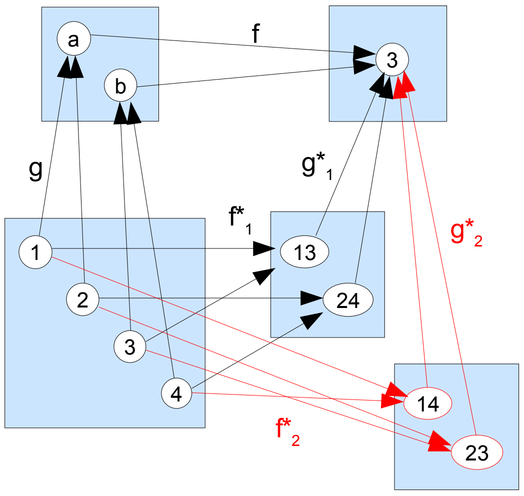

*Fig. 4: Ambiguous pullback complement in **SET***

Example 2 (Ambiguous Pullback Complements in Set). Let g:{1,2,3,4}→{a,b} be given by g(1)=a, g(2)=a, g(3)=b, g(4)=b and f:{a,b}→{3} be the constant function as in Fig. 4. There are two pullback complements:

1. ({13,24}, f^*_1:{1,2,3,4}→{13,24}, g^*_1: {13,24}→{3}) with f^*_1(1)=13, f^*_1(2)=24, f^*_1(3)=13, and f^*_1(4)=24.
2. ({14,23}, f^*_2:{1,2,3,4}→{14,23}, g^*_2: {13,24}→{3}) with f^*_2(1)=14, f^*_2(2)=23, f^*_2(3)=23, and f^*_2(4)=14.

Obviously, {13,24} and {14,23} are isomorphic. But no isomorphism i:{13,24}→{14,23} translates f^*_1 to f^*_2, in the sense: $i \circ f^*_1 = f^*_2$. Hence, we have isomorphic pullback complement objects. But the induced morphisms are ambiguous since they cannot be compared by the existing isomorphisms. □

This type of ambiguity cannot be accepted in our context, since the morphisms represent the transition of the data from the old to the new model. 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*. These properties shall single out a unique choice for the pullback complement extension of g.

These examples provide the motivation for the following definitions:

Definition 3 (Graph). The category **G** of *graphs* is the algebraic category w. r. t. the signature:

Sorts: O(bject)

Opns: s(ource), t(arget): O → O.

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 s(n) = n = t(n). Graphs and graph morphisms of this type provide more flexibility in the refactoring/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 s(e) = x and t(e) = y to z, too. E.g. in Fig. 1, *l(Unit)* = *l(Dept.)* = *Dept.* and the edge between them is mapped to the node *Dept.* as well.

Definition 4 (Component Graph). A *component graph* g: $G \rightarrow \underline{G}$ is a morphism in **G**. A *component graph morphism* α: (g: $G \rightarrow \underline{G}$) → ($h$: $H \rightarrow \underline{H}$) is a pair ($\alpha$: $G \rightarrow H$, $\underline{\alpha}$: $\underline{G} \rightarrow \underline{H}$) such that the resulting square commutes, i.e., $\underline{\alpha} \circ g = h \circ \alpha$. The comma category **CG** consists of all component graphs and all morphisms between them.

If not otherwise stated, we just write g for a component graph g: $G \rightarrow \underline{G}$. Note that G is the underlying graph and g provides a decomposition of G into parts or components via the congruence kern*(g)*[5]. Thus for the carrier set G we have $G=\Sigma\{[x]_g : x\in G\}$ where $[]_g$ denotes congruence classes of kern*(g)*. We also note that congruence classes are not necessarily subgraphs of G as can easily be seen in component graphs id: $G \to G$ where G contains edges.

The additional component structure on graphs provides means to distinguish typings from refactorings. In a typing, we require that all components are instantiated completely in a 1:1 manner. In a refactoring we allow identification of objects if and only if they belong to the same component. Hence, refactorings map components injectively and typings map objects within components bijectively.

Note that **CG** has all limits and that pullbacks in **CG** can be constructed component-wise.

Definition 5 (Typings, Refactorings, and Migrations).

A *typing* t: $g \to h$ is a **CG**-morphism if for each $x\in G$ the mapping $t:[x]_g\to[t(x)]_h$ considered as a **SET**-morphism is bijective.

A *refactoring* is a pair of morphisms $m\overset{l}{\leftarrow}k\overset{r}{\rightarrow}n$ in **CG** such that $\underline{l}$ and $\underline{r}$ are injective. The morphisms l and r are called *refactoring morphisms* in this case.

A refactoring $m\overset{l}{\leftarrow}k\overset{r}{\rightarrow}n$ and a typing $t:d\to m$ induce a *migration* from typing $t:d\to m$ to typing $u:e\to n$, if there is a diagram as depicted to the right that satisfies:

$$\begin{tikzcd}
m & k \arrow[l, "l"'] \arrow[r, "r"] & n \\
d \arrow[u, "t"] \arrow[ur, phantom, "(1)"] & f \arrow[l, "l'"] \arrow[r, "r'"'] \arrow[u, "v"'] \arrow[ur, phantom, "(2)"] & e \arrow[u, "u"']
\end{tikzcd}$$

1. (1) and (2) are pullbacks, and
2. r' is epimorphism.

The proof of the following proposition is straightforward and relies on the fact, that pullbacks preserve monomorphisms and isomorphisms.

Propositon 6 (Refactorings, Typings, and Pullbacks). If (n^*: $l \to g$, m^*: $l \to k$) is the pull-back of (n: $k \to h$, m: $g \to h$) in **CG**, then

1. m^* is a refactoring if m is,
2. n^* is a typing if n is, and
3. if n is injective on components, i.e., $\forall x,y\in K:(n(x)=n(y)\wedge k(x)=k(y))\Rightarrow x=y$, then the same property holds for n^*, i.e., $\forall x,y\in L:(n^*(x)=n^*(y)\wedge l(x)=l(y))\Rightarrow x=y$

Proposition 7 (Existence and Uniqueness of Migrations). Let $m\overset{l}{\leftarrow}k\overset{r}{\rightarrow}n$ be a refactoring and let
t: $d \to m$ be a typing. If r: $K \to N$ is an epimorphism, then:

1. There is a migration as defined in definition 5 and
2. The result of the migration is uniquely determined (up to isomorphism).

5 The relation kern(f) denotes the congruence that the morphism f induces on its domain, i.e., $(x, y) \in$ kern(f) iff $f(x) = f(y)$.

Proof. Subdiagram (1) can be constructed as a pullback. Thus (F, v, l') are unique up to isomorphism. The morphism v is a typing due to proposition 6. Having a typing v and a refactoring morphism r, we construct diagram (2), i.e., (E, r', u), as follows and depicted in Fig. 5:

If the component graph f is the morphism f: $F \rightarrow \underline{F}$, then

1. $\underline{r'}$ is the identity on $\underline{F}$,
2. $E = F/\equiv$ where $\equiv = \text{kern}(f) \cap \text{kern}(r \circ v)$
3. $r' = [\,]_{\equiv}$,
4. $\underline{u} = \underline{r} \circ \underline{v}$,
5. u is the unique morphism providing $u \circ r' = r \circ v$ which exists since $\text{kern}(r \circ v) \supseteq \equiv$, and
6. component graph e: $E \rightarrow \underline{E}$ is the morphism with $e \circ r' = f$ which exists since $\text{kern}(f) \supseteq \equiv$.

$$\begin{tikzcd}
\underline{K} \arrow[rrr, "\underline{r}"] & & & \underline{N} \\
& K \arrow[ul, "k"] \arrow[r, "r"] & N \arrow[ur, "n"'] & \\
& F \arrow[u, "v"] \arrow[r, "r'"] \arrow[dl, "f"'] & E \arrow[u, "u"] \arrow[dr, "e"] & \\
\underline{F} \arrow[uuu, "\underline{v}"] \arrow[rrr, "\underline{r'}"] & & & \underline{E} \arrow[uuu, "\underline{u}"']
\end{tikzcd}$$

Fig. 5: Constructing the right side of a refactoring

By construction $u \circ r' = r \circ v$ and r' is epimorphism. Since $\text{kern}(r') = \text{kern}(r \circ v)$ on each component and r is an epimorphism, u is bijective on components and thus a typing. And it is easy to show that (v, r') is pullback of (r, u) in **SET** and therefore in **CG**: if there is o such that $r(x(o)) = u(y(o))$, then choose $o' = r'^{-1}(y(o)) \cap v^{-1}(x(o))$. This is unique, since v is bijective on components and, by construction, r' folds on components only. This completes the proof of the first statement.

To prove the second statement, let (r^*: $F \rightarrow E'$, u^*: $E' \rightarrow N$) be any other completion with the required properties. It is easy to see, that the two pullback situations project to pullback situations in **SET** on each component. Here we have $u^* \circ r^* = u \circ r'$ where u^* and u are bijective. Hence kern(r^*) = kern(r') on each component of F. Because r is a refactoring, so are r' and r^* (see proposition 6) such that this property holds throughout F. Thus $E \cong E'$. □

Although the framework presented so far allows copying and gluing of objects *within the same component* only, it provides some nice features for our purposes of information system refactoring, as the following example demonstrates.

Example 8 (Association Redirection). In Fig. 1 we showed how to introduce a superclass *Unit*. Subsequently, one needs to check the references to *Department*-objects and redirect them to *Unit*-objects if necessary. To do this, consider the model refactoring in

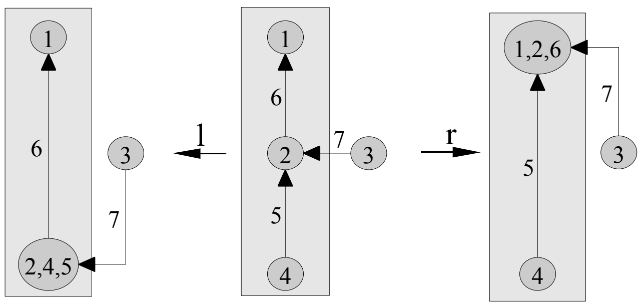

Fig. 6: Redirection of associations

Fig. 6[6]: All three graphs have 3 components; the non-trivial component in each graph (the component that has more than one element) is highlighted. Using this refactoring 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 as long as we stay in the same component. □

With these features, we should be able to handle all refactorings that are concerned with inheritance structures. Recall, that inheritance can be considered as 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*. 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 (Def. 5). It is not the case, that the *complete* inheritance tree of classes needs to be instantiated, if one class is. If there are concrete classes that possess subclasses, an object might instantiate a proper subpart of the complete inheritance tree of its class, only. Our approach is not able to handle those incomplete parts, since pullback complements do not always exist in these situations.

Example 9 (Missing Pullback Complement). Consider the reverse process as in example 8. An association to class “1,2,6” (a concrete superclass of “4”) shall be redirected to its subclass, see Fig. 7. We apply this rule to an instance of “1,2,6”, called “1,2,6' ”. The pullback on the left produces an intermediate object “2' ” in F. But we can easily deduce, that the right part is not able to complete the diagram to a double-pullback situation. This is mainly due to the fact that the non-trivial component in K is only partially instantiated in F (there is no “4”-object). For suppose, that such a pullback complement *r':* *F → E, u: E → N* exists. Then *r'(2')* is a preimage of “2,4,5” under *u*. This is only possible if there is a preimage of “4” in F. □

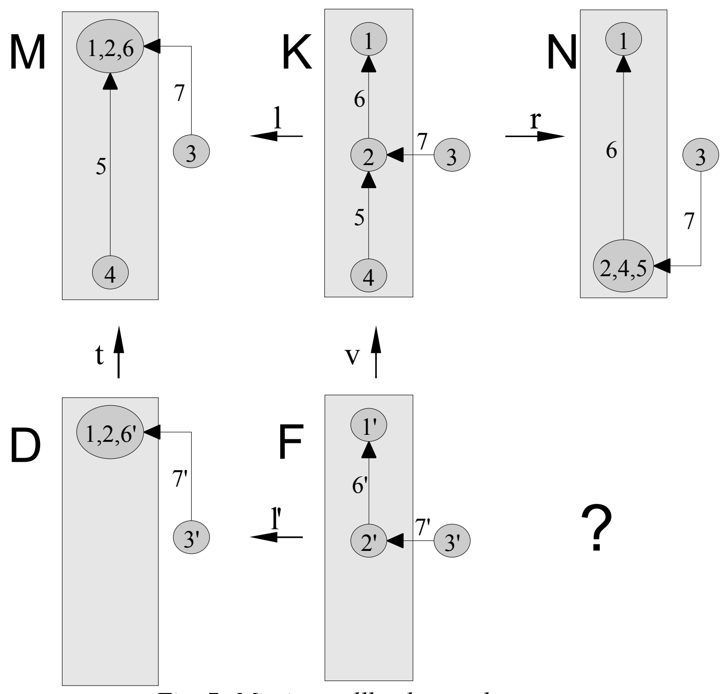

Fig. 7: Missing pullback complement

6 We indicate the model objects numerically to clarify the mappings.

We might 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 *refactoring 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.[7] With these additional arrangements, inheritance structures and the typical refactorings could be modeled.

But there are also disadvantages of this approach: the additional instantiations might cause a significant memory overhead. In this paper, we use a different approach, which omits this problem: In the next section, we slightly generalize our framework such that partial instantiations of components in the model are allowed. To achieve that, we do no longer require that the right-hand side of a migration is a pullback.

3. Partial Instantiation of Components

In this section, we relax the requirements for typings. *Weak typings* allow partial instantiations of model components, since they are injective on each component but need not be surjective.

Definition 10 (Weak Typing). A component graph morphism α: $(g: G \to \underline{G}) \to (h: H \to \underline{H})$ is a *weak typing* if it is injective on each component, i.e., $\alpha(x) = \alpha(y) \wedge g(x) = g(y) \Rightarrow x = y$.

Now we use the construction in the proof of proposition 7 to construct the right-hand side of a migration. This works for weak typings as well.

Construction 11 (Folding). Consider Fig. 8, where weak typing n and refactoring morphism m are given. We construct the *folding completion* of this situation as follows:

1. $m^* = ([]_\equiv, \mathrm{id}_{\underline{G}})$, where $\equiv \; = \mathrm{kern}(g) \cap \mathrm{kern}(m \circ n)$,
2. $n^* = (i, \underline{m} \circ \underline{n})$, where i is the unique morphism with $i \circ []_\equiv = m \circ n$, since $\mathrm{kern}(m \circ n) \supseteq \equiv$, and
3. the component graph j is the unique morphism with $j \circ []_\equiv = \mathrm{id} \circ g$, since $\mathrm{kern}(g) \supseteq \equiv$.

$$\begin{tikzcd}
\underline{H} \arrow[rrr, "\underline{m}"] & & & \underline{K} \\
& H \arrow[lu, "h"] \arrow[r, "m"] & K \arrow[ru, "k"] & \\
& G \arrow[u, "n"] \arrow[r, "{[]_\equiv}"] \arrow[ld, "g"] & G/_\equiv \arrow[u, "i"] \arrow[rd, "j"] & \\
\underline{G} \arrow[uuu, "\underline{n}"] \arrow[rrr, "id"] & & & \underline{G} \arrow[uuu, "\underline{m} \circ \underline{n}"']
\end{tikzcd}$$

Fig. 8: Construction of a Folding

m^* is a **CG**-morphism by construction. Moreover, we obtain $k \circ i = \underline{m} \circ \underline{n} \circ j$, since $[]_\equiv$ is epi and $k \circ i \circ []_\equiv = k \circ m \circ n = \underline{m} \circ \underline{n} \circ g = \underline{m} \circ \underline{n} \circ j \circ []_\equiv$. Thus, n^* is a **CG**-morphism, too. □

Lemma 12 (Folding). If (m^*, n^*) is folding of (m, n), m^* is refactoring morphism and n^* is typing.

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

Proof. The first part is obvious, since $\underline{m^*}$ has been constructed as the identity which is a mono.

For the proof of the second statement let $i(x) = i(y)$ and $j(x) = j(y)$. Now consider arbitrary preimages x' and y' for x and y wrt. $[]_\equiv$, i.e., $[x']_\equiv = x$ and $[y']_\equiv = y$. Since $j \circ []_\equiv = id \circ g$, we conclude $g(x') = g(y')$. Since $i \circ []_\equiv = m \circ n$, it follows that $m(n(x')) = m(n(y'))$.

Thus, $(x', y') \in \mathrm{kern}(g) \cap \mathrm{kern}(m \circ n)$, which means that $[x']_\equiv = [y']_\equiv$. Hence, $x = y$. □

Folding diagrams possess an interesting universal property as the following proposition shows.

Proposition 13 (Initiality of Foldings). Let the pair of morphisms (m^*: $g \to f$, n^*: $f \to k$) be the folding of a weak typing n: $g \to h$ and a refactoring morphism m: $h \to k$ as it is constructed in construction 11. Then for every triple of morphisms (w: $g \to b$, t: $b \to a$, v: $k \to a$) such that t is weak typing and $t \circ w = v \circ m \circ n$, there is a unique morphism u: $f \to b$ with $t \circ u = v \circ n^*$ and $u \circ m^* = w$.

Proof. Let the folding be given as in Fig. 8. We set $\underline{u} = \underline{w}$ and get immediately (1) $\underline{u} \circ \underline{m}^* = \underline{u} \circ \mathrm{id} = \underline{u} = \underline{w}$. We show that $\equiv \subseteq \mathrm{kern}(w)$. Let $m(n(x)) = m(n(y))$ and $g(x) = g(y)$. It follows $t(w(x)) = v(m(n(x))) = v(m(n(y))) = t(w(y))$ and $b(w(x)) = \underline{w}(g(x)) = \underline{w}(g(y)) = b(w(y))$. Since t is weak typing, we get $w(x) = w(y)$ as desired. Now there is a unique u: $G/_\equiv \to B$ with (2) $u \circ []_\equiv = u \circ m^* = w$. Since $b \circ u \circ []_\equiv = b \circ w = \underline{w} \circ g = \underline{w} \circ j \circ []_\equiv = \underline{u} \circ j \circ []_\equiv$, we can conclude (3) $b \circ u = \underline{u} \circ j$. And $t \circ u \circ []_\equiv = t \circ w = v \circ m \circ n = v \circ i \circ []_\equiv$ provides (4) $t \circ u = v \circ i = v \circ n^*$. Finally, we also have (5) $\underline{t} \circ \underline{u} = \underline{v} \circ \underline{m} \circ \underline{n} = \underline{v} \circ \underline{n}^*$. □

Proposition 13 characterizes foldings up to isomorphism. In the following, we say that a diagram is an abstract folding if it has the property of proposition 13:

Definition 14 (Abstract Folding). As depicted in Fig. 9, a pair (m^*: $g \to f$, n^*: $f \to k$) consisting of a refactoring morphism m^* and a weak typing n^* is the *abstract folding* of a weak typing n: $g \to h$ and a refactoring morphism m: $h \to k$ if (1) $n^* \circ m^* = m \circ n$ and (2) for every triple (w: g $\to b$, t: $b \to a$, v: $k \to a$) such that t is weak typing and $t \circ w = v \circ m \circ n$, there is a unique morphism u: $f \to b$ with $t \circ u = v \circ n^*$ and $u \circ m^* = w$.

$$\begin{tikzcd}
h \arrow[r, "m"] & k \arrow[r, "v"] & a \\
g \arrow[u, "n"] \arrow[r, "m^*"] \arrow[rrd, bend right, "w"'] & f \arrow[u, "n^*"'] \arrow[rd, "u!"] & \\
 & & b \arrow[uu, bend right, "t"']
\end{tikzcd}$$

Fig. 9: Abstract Folding

Corollary 15 (Uniqueness of Abstract Foldings). Two abstract foldings of a weak typing n: $g \to h$ and a refactoring morphism m: $h \to k$ coincide up to isomorphism. Hence (m^*: $g \to f$, n^*: $f \to k$) is the abstract folding if and only if the statement

$$m^* \text{ is epimorphism and } m^*(x) = m^*(y) \Leftrightarrow (g(x) = g(y) \wedge m(n(x)) = m(n(y)))$$

holds.

Proof. Direct consequence of Definition 14 and the fact that the first folding compares to the second and vice versa. Therefore, we get two morphisms between the two foldings, which must be inverse morphisms, because their composition coincide with the identity on the folding objects (unique morphism from a folding to itself). □

Abstract foldings enjoy the same composition and decomposition properties as pushouts or pullbacks.

Proposition 16 (Composition and Decomposition of Abstract Foldings).

Consider the situation depicted in the diagram below[8].

1. If the squares (1) and (2) are abstract foldings, then the rectangle (1) + (2) is an abstract folding.[9]
2. If the rectangle (1) + (2) and the square (1) are abstract foldings, then (2) is an abstract folding.
3. If typing e and refactoring h is the abstract folding of typing c and refactoring $b \circ a$, it can be decomposed into two foldings as in the diagram on the right, where $h = g \circ f$, if the underlying category has all abstract foldings.

$$\begin{tikzcd}
A \arrow[r, "a"] & B \arrow[r, "b"] & C \\
D \arrow[u, "c"] \arrow[r, "f"] \arrow[ur, phantom, "(1)"] & E \arrow[u, "d"] \arrow[r, "g"] \arrow[ur, phantom, "(2)"] & F \arrow[u, "e"]
\end{tikzcd}$$

Proof.

(1) Let morphisms v, t, w, be given such that t is typing and $t \circ w = v \circ b \circ a \circ c$. Since (1) is abstract folding, we get u_1 such that $u_1 \circ f = w$ and $t \circ u_1 = v \circ b \circ d$. Now u_1, t and v compare to (2) and we get u_2 with $u_2 \circ g = u_1$ and $t \circ u_2 = v \circ e$. Substituting $u_2 \circ g = u_1$ in $u_1 \circ f = w$ provides $u_2 \circ g \circ f = w$. For the proof of uniqueness, let morphism u_3 be given such that $u_3 \circ g \circ f = w$ and $t \circ u_3 = v \circ e$. Then $u_3 \circ g \circ f = u_2 \circ g \circ f$ and $t \circ u_3 \circ g = v \circ b \circ d = t \circ u_2 \circ g$ hold. We obtain $u_3 \circ g = u_2 \circ g$, since (1) is abstract folding. But this implies $u_3 = u_2$, since (2) is abstract folding.

(2) Let v, t, w, be given such that t is typing and $t \circ w = v \circ b \circ d$. It follows $t \circ w \circ f = v \circ b \circ a \circ c$. Since (1)+(2) is abstract folding, there is u such that $u \circ g \circ f = w \circ f$ and $t \circ u = v \circ e$. We also have $t \circ u \circ g = v \circ e \circ g = v \circ b \circ d$. Since (1) is abstract folding, we get $u \circ g = w$. Uniqueness follows from the uniqueness of u for (1)+(2).

(3) If there are all abstract foldings, we can construct (d, f) as a folding, which provides diagram (1). The morphism g is obtained as the unique completion of the diagram from the folding (1). That diagram (2) is an abstract folding follows from (2) of this proposition. □

Definition 17 (Generalized Migration). A refactoring $m \xleftarrow{l} k \xrightarrow{r} n$ and a weak typing $t : d \to m$ induce a *generalized migration* from $t : d \to m$ to weak typing $u : e \to n$, if there is a diagram as depicted to the right that satisfies:

1. Subdiagram (1) is pullback and
2. Subdiagram (2) is abstract folding.

$$\begin{tikzcd}
m & k \arrow[l, "l"'] \arrow[r, "r"] & n \\
d \arrow[u, "t"] & f \arrow[l, "l'"] \arrow[r, "r'"'] \arrow[u, "v"'] \arrow[ul, phantom, "(1)"] \arrow[ur, phantom, "(2)"] & e \arrow[u, "u"']
\end{tikzcd}$$

8 Here, for the sake of readability, **CG**-objects are presented in capital letters.
9 (1)+(2) consists of the morphisms $b \circ a, e, g \circ f$, and c.

Theorem 18 (Existence and Uniqueness of Generalized Migrations). Given a weak typing $t: D \to M$ and refactoring $M \xleftarrow{l} K \xrightarrow{r} N$ for the model of *t*, there is an induced migration and the result of the migration is unique up to isomorphism.

Proof. Direct consequence of (1) the existence and uniqueness of pullbacks in **CG**, (2) the fact that pullbacks in **CG** preserve weak typings (proposition 6, 3.), and (3) the existence and uniqueness of abstract foldings in **CG** (Construction 11 and Corollary 15). □

Theorem 18 justifies that we write *R*(*t*) for the result typing of a migration from a typing $t: D \to M$ using a refactoring *R* = $M \xleftarrow{l} K \xrightarrow{r} N$. Fig. 10 shows the generalized migration that we searched for in Fig. 7.

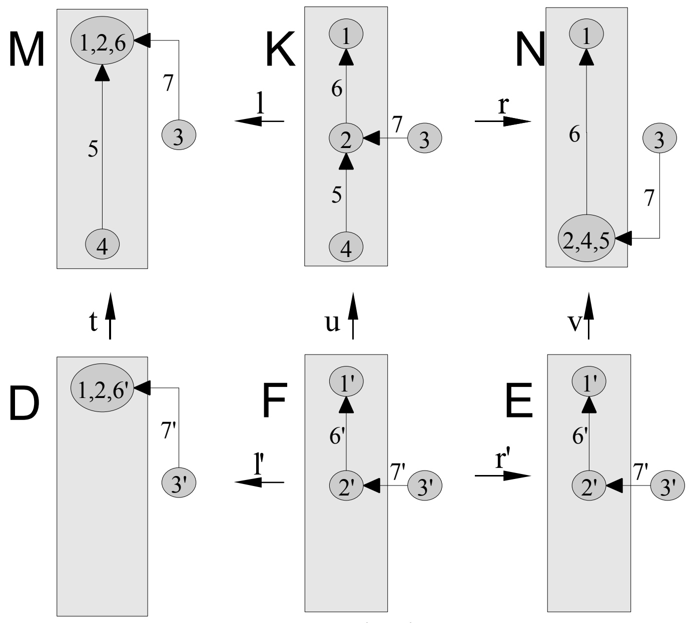

Fig. 10: A generalized migration

4. Sequential Composition

In this section, we show that there is a natural sequential composition $R_2 \circ R_1$ of refactorings R_1 and R_2 and that applying a sequential composition to a weak typing *t* provides exactly the same result as the sequence of first applying R_1 to *t* and second R_2 to $R_1(t)$, i.e., $R_2 \circ R_1(t) = R_2(R_1(t))$.

Definition 19 (Sequential Composition of refactorings). 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 refactorings $R_1 = (l_1: K \to M, r_1: K \to N)$ and $R_2 = (l_2: H \to N, r_2: H \to P)$ is defined with the help of the pullback object $(p_1: J \to K, p_2: J \to H)$ of r_1 and l_2 as depicted in Fig. 11.

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

Fig. 11: Sequential composition

Note that the sequential composition is well-defined due to proposition 6, 1. and the fact that the composition of two refactoring morphisms is a refactoring morphism again.

In order to prove our main theorem, i.e., $R_2 \circ R_1(t) = R_2(R_1(t))$, we need the following technical lemma.

Lemma 20 (Pullback Cubes Preserve Abstract Foldings). Consider the commuting diagram in **CG** below[10]. If the pair of morphisms (*i*, *q*) is the abstract folding of the morphism pair (*r*, *m*),

10 Here, we depict **CG**-objects as arrows with filled tip.

the pair (*p*, *t*) is the pullback of the pair (*s, q),* and (*j*, *v*) is the pullback of (*t*, *i*), then the pair of morphisms (*j*, *p*) is the abstract folding of *(n, k)*.

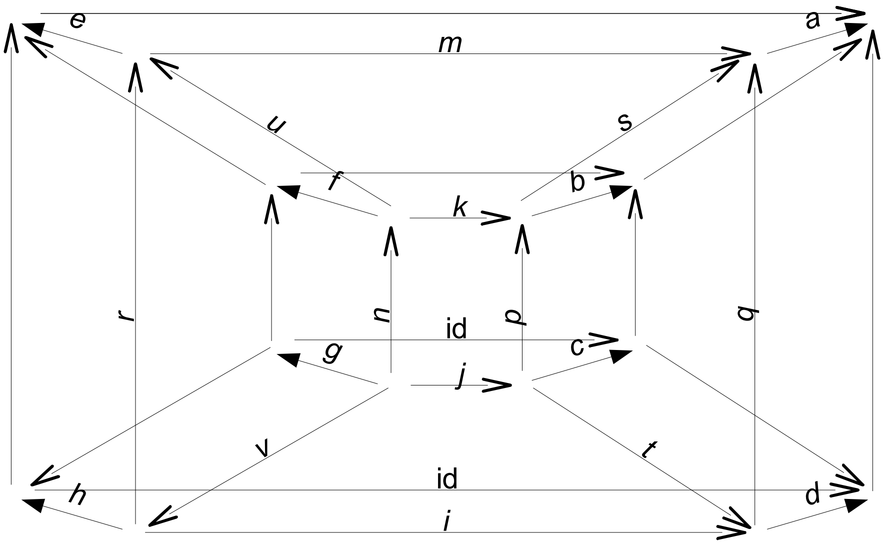

Fig. 12: Pullback Cube

Proof. The assumptions of the lemma provide that $\underline{i}$ is the identity. Since pullbacks preserve isomorphisms, we can set $\underline{j}$ = id without loss of generality. Because the bottom face is a pullback and *i* is an epimorphism (see Corollary 15), *j* is an epimorphism as well. Again, from Corollary 15 we deduce that it suffices to show, that

$$j(x) = j(y) \Leftrightarrow [k(n(x)) = k(n(y)) \wedge g(x) = g(y)]$$

holds for all $x, y \in G$.

"⇒": (1) $j(x) = j(y) \Rightarrow p(j(x)) = p(j(y)) \Rightarrow k(n(x)) = k(n(y))$

(2) $j(x) = j(y) \Rightarrow c(j(x)) = c(j(y)) \Rightarrow \mathrm{id}(g(x)) = \mathrm{id}(g(y)) \Rightarrow g(x) = g(y)$

"⇐": Let $k(n(x)) = k(n(y)) \wedge g(x) = g(y)$ be given. Since (*p*, *t*) is pullback, it is sufficient to show: (3) $t(j(x)) = t(j(y))$ ∧ (4) $p(j(x)) = p(j(y))$:

(3) (a) $g(x) = g(y) \Rightarrow \underline{v}(g(x)) = \underline{v}(g(y)) \Rightarrow h(v(x)) = h(v(y))$.

(b) $k(n(x)) = k(n(y)) \Rightarrow s(k(n(x))) = s(k(n(y))) \Rightarrow m(r(v(x))) = m(r(v(y)))$.

Since (*q*, *i*) is abstract folding, it follows from Corollary 15, (a) and (b) that $i(v(x)) = i(v(y))$, which provides $t(j(x)) = t(j(y))$, because $t \circ j = i \circ v$.

(4) $k(n(x)) = k(n(y)) \Rightarrow p(j(x)) = p(j(y))$. □

Theorem 21 (Sequential Composition). If $R_2(R_1(t))$ for two refactorings R_1 and R_2 is defined, we have $R_2 \circ R_1(t) = R_2(R_1(t))$.

Proof. Consider the following diagram, which depicts $R_2(R_1(t))$. This migration sequence is given by the four squares (1) MKFD, (2) KNEF, (3) NHCE, and (4) HPBC. (1) and (3) are pullbacks and (2) and (4) are abstract foldings. The additional material in the diagram is defined as follows: The pair of morphisms (p_1, p_2) is given as a pullback of r_1 and l_2, compare construction of $R_2 \circ R_1$ in definition 19. We write (5) for the resulting square NKJH. We construct (p_1',p_2') as pullback of (r_1',l_2'). We write (6) for the resulting square EFIC. The morphism u_2 is the universal completion of the diagram into the pullback object J. Now, the square (7) KJIF is pullback as well. This is due to the fact that (3)+(6) is pullback[11], (3)+(6) = (5)+(7), and (5) is pullback[12]. The square (8) JICH is abstract folding due to Lemma 20. Now diagram (1)+(7) is pullback, since pullbacks compose. It is the left-hand side of the migration induced by $R_2 \circ R_1$. Diagram (8)+(4) is abstract folding, since abstract foldings compose (compare Proposition 16, 1.). It is the right-hand side of the migration induced by $R_2 \circ R_1$. This together shows that $R_2 \circ R_1$ migrates t to w as well. □

$$\begin{tikzcd}
 & & & J \arrow[dll, "p_1"'] \arrow[drr, "p_2"] & & & \\
M & K \arrow[l, "l_1"'] \arrow[drrr, "r_1"] & & & & H \arrow[r, "r_2"] \arrow[dl, "l_2"] & P \\
 & & & & N & & \\
 & & & I \arrow[uuu, "u_2"] \arrow[dll, "p_1'"'] \arrow[drr, "p_2'"] & & & \\
D \arrow[uuu, "t"] & F \arrow[l, "l_1'"] \arrow[uuu, "v_1"] \arrow[drrr, "r_1'"'] & & & & C \arrow[uuu, "v_2"] \arrow[r, "r_2'"'] \arrow[dl, "l_2'"] & B \arrow[uuu, "w"'] \\
 & & & & E \arrow[uuu, "u_1"] & &
\end{tikzcd}$$

Fig. 13: Migration sequence

With Theorem 21 we are, on the one hand, able to compose long refactoring sequences into one single refactoring, which can capture the effect of the whole sequence. On the other hand, we can decompose complex refactorings into a composition of simpler ones.

5. General Framework

The whole approach presented above is almost independent from the underlying category of graphs resp. component graphs. What we need for the existence and uniqueness of migrations is the existence of pullbacks and abstract foldings. For the results concerning sequential composition, we need the cube lemma 20, i.e., that pullbacks "pull back" abstract foldings. Thus, we can present our requirements for a category to provide the infrastructure for unique migrations and sequential compositions as follows:

An abstract *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 *refactoring morphisms*. The system (C, T, R) is subject to the following requirements:

(1) C has all pullbacks

11 Composition property of pullbacks.
12 Decomposition property of pullbacks.

(2) C has abstract foldings for all pairs of morphisms (f: A $\rightarrow$ B $\in$ T, g: B $\rightarrow$ C $\in$ R).

(3) Pullbacks in C preserve morphisms of T and of R.

(4) In each cube with corners K, N, H, J, F, E, C, and I, as it is depicted in Fig. 13, the square JICH is an abstract folding if KNEF is abstract folding and the squares IFEC and NECH are pullbacks.

Since abstract foldings are a generalization of surjective pullback complements[13], the framework presented in section 2 fits into this setting as well. Another instance is given by simple graphs, arbitrary morphisms as typings and injective morphisms as refactoring morphisms. Here we can use surjective pullback complements as abstract foldings as well [15].

6. Conclusion

We propose formalizations of aspects in the process of refactoring information systems. The power of our attempt is that a model refactoring can uniquely and automatically be extended to the instance level. In contrast to other more practical solutions, we can prove correctness of our approach. The framework is described using abstract notions from category theory.

With a strong assumption to the typing morphisms we can generalize a migration to a double-pullback diagram. As a first step, it is possible to handle addition, renaming, and removal of model objects. The investigation under which conditions folding and unfolding is possible, leads to a model structure where one had to restrict to 1:1 associations on certain components. A refactoring morphism may fold or unfold on these components, only. In a second step we showed that these settings are correct as well.

However, object trees of inheritance structures are, in general, not completely instantiated. To treat this case in a similar way, we have to weaken the assumptions on the type mappings. But weak typings do not always lead to double-pullback constructions. Thus, this third step requires a generalization of pullback complements. We introduced abstract foldings that enjoy some of the well-known properties of pullbacks and pushouts. Abstract foldings are initial in a reasonable context, which reveales a uniqueness statement of generalized migrations and prepares a statement on the composition of refactorings.

Composing migrations into larger projects and decomposing migrations into smaller steps leads to the question if there is a minimal set of *atomic* refactorings, from which each refactoring can be constructed by sequential composition. This might be an interesting topic for future research as well as the question, under which conditions refactorings are parallel or sequential independent and can be performed concurrently. These results are valuable for tools that produce migrations on the basis of the construction of pullbacks and abstract foldings.

Finally, in a forth step, we describe a way of integrating refactoring and migration procedures in a more general framework that abstracts away from the underlying category. We define requirements that are the basis for a generalized system. These requirements are very similar to the axioms for adhesive categories in [14]. It is up to future research to investigate if both frameworks can be seen as two instances of an even more general system.

13 pullback complements such that the morphism into the complement is surjective

References

1 Havey, M.: Essential Business Process Modeling. O'Reilly (2005)
2 Martin, R. C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall (2002)
3 Beck, K.: Extreme Programming Explained. Addison Wesley (2000)
4 Beck, K.: Test-driven Development by Example. Addison-Wesley (2002)
5 Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
6 Kerievsky, J.: Refactoring to Patterns. Addison-Wesley (2004)
7 D'Anjou, J et al: The Java Developer's Guide to Eclipse. Addison-Wesley (2005)
8 Ambler, S. W.: Agile Database Techniques. Wiley (2003)
9 Ambler, S. W.: Refactoring Databases : Evolutionary Database Design. Addison-Wesley (2006)
10 Hainaut, J.-L.: Introduction to database reverse engineering. LIBD Publish. (2002)
11 Bauer, Ch., King, G.: Hibernate in Action. Manning Publications (2004)
12 Löwe, M.: Evolution Patterns – A Graphical Framework for Software Redesign. Proceedings ISAS'99 (1999)
13 Adamek, J., Herrlich, H., Strecker G. E.: Abstract and Concrete Categories – The Joy of Cats. (2004) [http://katmat.math.uni-bremen.de/acc/acc.pdf]
14 Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
15 Löwe, M., König, H., Peters, M., Schulz, Ch.: A Formal Framework for Information System Refactorization. Proceedings WMSCI 2006, Vol. 1, 75-80 (2006)
16 Meisen, J.: Pullbacks in Regular Categories. Canad. Math. Bull. Vol.16(2) (1973)
17 Bauderon, M., Jacquet, H.: Pullback as a generic graph rewriting mechanism. Applied Categorical Structures Vol.9(1) (2001)