Refactoring Object-Oriented Systems

Christoph Schulz, Michael Löwe, and Harald König

Abstract. Refactoring of information systems is hard, for two reasons. On the one hand, large databases exist which have to be adjusted. On the other hand, many programs access that data. These programs all have to be migrated in a consistent manner such that their semantics does not change. It cannot be relied upon, however, that no running processes exist during such a migration. Consequently, a refactoring of an information system needs to take care of the migration of data, programs, *and* processes. This paper introduces a model for complete object-oriented systems, describing the schema level with classes, associations, operations, and inheritance as well as the instance level with objects, links, methods, and messages. Methods are expressed by special double-pushout graph transformations. Homomorphisms are used for the typing of the instance level as well as for the description of refactorings which specify the addition, folding, and unfolding of schema elements. Finally, a categorial framework is presented which allows to derive instance migrations from schema transformations in such a way that programs and processes to the old schema are correctly migrated into programs and processes to the new schema.

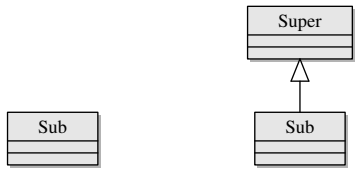
1 Introduction

During the engineering and use of information systems, data and software undergo many modifications. These modifications can be divided into two categories. The first category contains all modifications that have a direct and externally visible impact on the functionality of the software or on the information content of the database. The second category consists of modifications which only *prepare* modifications of the first category and which, by themselves, do not lead to changes in the behaviour of the software or in the meaning of the data under transformation. Modifications of the second category are called "refactorings" [1]. They provide a major method to quickly adapt software to constantly changing requirements.

Refactorings are expected to be applied multiple times in different but similar situations. This is comparable to *design patterns* in software engineering which have emerged in the last twenty years [2, 3]. Consequently, a suitably general specification of a refactoring is necessary. This, however, requires a certain level of *abstraction* for the software and the data to be transformed. Such an abstraction is often called *schema* or *model* and describes important structural aspects of the data and software, which are *instances* of, or *typed* in, this schema. Today, the "object-oriented view of life" dominates the field of software engineering. Therefore, models are typically object-oriented and try to capture the structure by grouping similar objects into *classes* and describing relations between them by various types of *associations*.

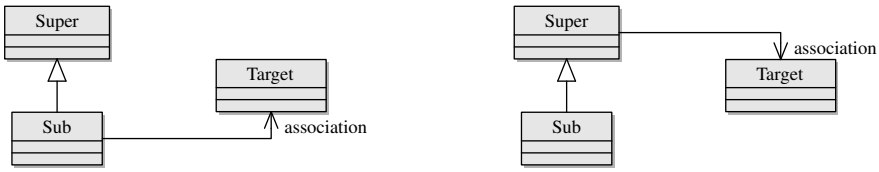
F. Drewes, A. Habel, B. Hoffmann, D. Plump (Eds.): Manipulation of Graphs, Algebras and Pictures. Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday, pp. 321–340, 2009.

Two typical object-oriented refactorings are "Introduce a new superclass" (Fig. 1) and "Move the origin of an association from a subclass to a superclass", as shown in Fig. 2. A combined application of these two refactorings on the schema in Fig. 3a could be used to prepare the model for an extension by an additional subclass of *Customer*, e. g. *CorporateCustomer* (Fig. 3b and 3c).[1]



(a) before (b) after

Fig. 1: Refactoring "Introduce a new superclass"



(a) before (b) after

Fig. 2: Refactoring "Move the origin of an association from a subclass to a superclass"

It is important to consider the *consequences* of a refactoring. Obviously, the more general the structures are which are about to be transformed, the more instances are likely to be affected. Changing a data schema may not only require the data typed in this schema to be adjusted, but may also affect the software which uses the schema structures to access and manipulate the data. Changing a software model may have no consequences on the data but will probably influence programs (which can be considered *implementations* of the software model) and processes (which are programs under execution). We call the instance changes that follow from a model refactoring the *migration* induced by that refactoring. For the time being, little has been written about how refactoring data models results in migrations of dependent programs, and even less has been written

[1] All class diagrams are specified in the UML [4].

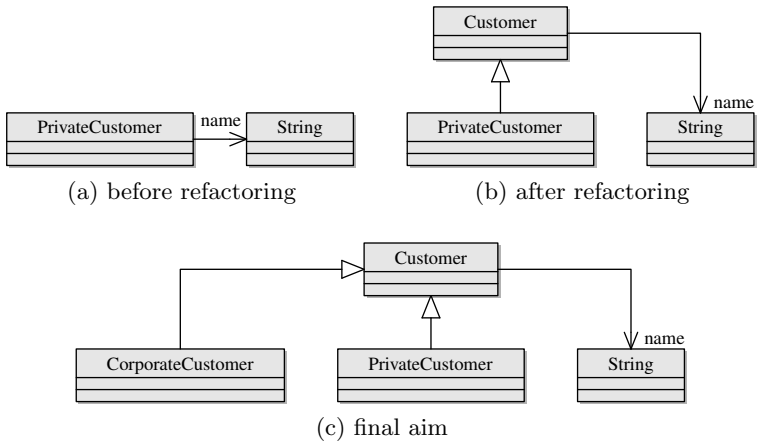


Fig. 3: Refactoring an exemplary object-oriented model

about refactoring and induced migration of *whole systems*, which we define to consist of data, programs, and processes, all typed in the same schema.[2]

This paper contributes to this topic by providing a graph-like mathematical model which allows to specify object-oriented systems as well as schema refactorings. To describe data together with their schema, *graph structures* conforming to the model are used. Nodes of such graph structures represent classes (schema) or objects (instance), edges represent associations (schema) or links (instance). *Homomorphisms* between such graph structures express *typings*, (parts of) *refactorings*, and *migrations*.

These graph structures can be used to describe software models and processes, as well: An operation is simply a special node within the graph structure representing the schema, and edges originating from an operation constitute parameters. Analogously, at the instance level, messages and arguments are special nodes and edges, which are typed in operations and parameters at the schema level by an appropriate homomorphism. In the mathematical description of the model, the data and software constructs are separated through the use of special *predicates*.

Programs are somewhat different, as they do not specify a single state but rather a state *transition* which is performed when the program is executed. In this paper, programs are considered to consist of a (possibly large) set of methods, where each method describes a single state transition. Each such transition speficies how a message of a certain type is processed when all necessary preconditions are met; examples are assignments, method calls, or evaluation of expressions. As program states are described by (parts of) graph structures at the instance level, it follows that state transitions can be adequately specified by

[2] See [5] and especially the bibliography contained therein for a general overview on software refactoring.

the use of *graph structure transformations*. This paper chooses the DPO approach for describing and applying graph structure transformations[3]. Consequently, a method is represented by a span of homomorphisms, and applying a method to a given program state is computed by two pushout diagrams.

Results of category theory are used to compute induced migrations from schema refactorings. It will be shown, however, that certain restrictions must be obeyed in order to guarantee reasonable results. Fortunately, these restrictions are met by the practical examples.

The paper is organized as follows. Section 3 incrementally introduces a graph structure specification *MP* with positive Horn formulas which constitutes the foundation of the mathematical description of data and software. The category $\mathbf{Alg}(MP)$ of all *MP*-systems and *MP*-homomorphisms, as well as the (sub-)categories $\mathbf{Alg}(MP)\downarrow S$ and $\mathbf{Sys}(S)$ with a fixed schema S, represent the universe of discourse for the following sections. Section 4 explains how methods are represented as DPO rules and introduces requirements that are necessary to use DPO graph structure transformations successfully in the categories mentioned above. Section 5 addresses the migration of data and processes. Section 6 discusses the migration of programs and contains the main result of this paper, namely that the migration of methods preserves their semantics for new processes as well as for old processes reviewed under the transformed schema. Section 7 outlines three main directions for future research.

Due to lack of space, this paper does not contain any proofs. All the proofs can be found in [7].

2 Related Work

There exist approaches for modelling programs as algebraic graph transformation rules [8–10]. However, they fail in various ways to be suitable for our purposes. The approach in [8] does not support inheritance. Furthermore, program execution is "destructive", i. e., repetitive control flow constructs as loops cannot be modelled directly but have to be simulated through recursion, a work-around which is not necessary in our approach as the control flow structures are not modified by program execution. The approach presented in [9, 10] does not have a notion of a schema in which programs and processes are typed. This missing link makes it hard if not impossible to compute induced migrations for programs and processes when the data schema is changed. Finally, both approaches consider objects to be opaque, whereas in our approach, each object is decomposed into parts called "particles" which reflect the class hierarchy. This rich object structure makes it possible to type the instance level in a schema without resorting to special typing morphisms or type graph flattening as proposed in [6, 11, 12]. Finally, our approach is unique in the respect that it combines a program and process model with a model for schema transformations and induced migrations.

[3] DPO stands for "Double Pushout"; the approach is presented in e. g. [6].

3 Models and Instances

The schema and the instance level of object-oriented systems are modelled by systems wrt. an extended specification.[4] An *extended specification* $Spec = (\Sigma, H(X))$ is an extended signature together with a set of positive Horn formulas $H(X)$ over a set of variables X. An *extended signature* $\Sigma = (S, OP, P)$ consists of a set of *sorts* S, a family of *operation symbols* $OP = (OP_{w,s})_{w \in S^*, s \in S}$, and a family of *predicates* $P = (P_w)_{w \in S^*}$ such that $=_s \in P_{s\ s}$ for each sort $s \in S$. A *system* A wrt. an extended signature $\Sigma = (S, OP, P)$, short Σ-system, consists of a family of *carrier sets* $(A_s)_{s \in S}$, a family of *operations* $(op^A \colon A_w \to A_s)_{w \in S^*, s \in S, op \in OP_{w,s}}$, and a family of *relations* $(p^A \subseteq A_w)_{w \in S^*, p \in P_w}$ such that $=_s^A \subseteq A_s \times A_s$ is the diagonal relation for each sort s.[5] A *system* A wrt. an extended specification $Spec = (\Sigma, H(X))$ is a Σ-system such that all axioms are valid in A. A *Σ-homomorphism* $h \colon A \to B$ between two Σ-systems A und B wrt. an extended signature $\Sigma = (S, OP, P)$ is a family of mappings $(h_s \colon A_s \to B_s)_{s \in S}$, such that the mappings are compatible with the operations and relations, i. e., $h_s \circ op^A = op^B \circ h_w$ for all operation symbols $op \colon w \to s$ and $h_w(p^A) \subseteq p^B$ for all predicates $p \colon w$ where $w = s_1 s_2 \dots s_n \in S^*$.[6] Each Σ-homomorphism $h \colon A \to B$ between two *Spec*-systems A und B wrt. an extended specification $Spec = (\Sigma, H(X))$ is called a *Spec*-homomorphism.

The (first) model version for classes and associations is just graphs as depicted in Fig. 4. Nodes correspond to classes and edges correspond to associations. In Fig. 5a, an exemplary UML schema is presented. The underlying graph for this schema is shown in Fig. 5b. Figure 5c illustrates the resulting *Graph* system.

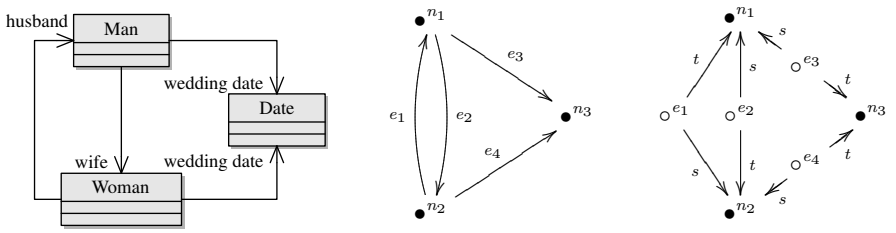
Graph =
sorts
N (nodes)
E (edges)
opns
$s \colon E \to N$ (source node of an edge)
$t \colon E \to N$ (target node of an edge)

Fig. 4: The *Graph* signature

An instance of a given schema S is represented as a system I wrt. the same signature, together with a typing homomorphism $type \colon I \to S$. At the instance level, nodes represent objects and edges constitute links.

The next model version provides the possibility to model inheritance relations between classes by an additional binary predicate *under*: If, in a system S, a

[4] See [13] for the special case when signatures consist of only one sort.
[5] Given $w = s_1 s_2 \dots s_n$, A_w is an abbreviation for the product set $A_{s_1} \times A_{s_2} \times \dots \times A_{s_n}$.
[6] Given $w = s_1 s_2 \dots s_n$, $h_w(x_1, x_2, \dots, x_n)$ is an short-hand notation for the term tuple $(h_{s_1}(x_1), h_{s_2}(x_2), \dots, h_{s_n}(x_n))$.



(a) UML schema (b) Representing graph (c) Representing algebra

Fig. 5: A system for the *Graph* signature

class A is "under" a class B, i. e., if it is a subclass of B, then the relation $under^S$ contains the pair (A, B). The specification MP_1 is shown in Fig. 6.[7] As inheritance is hierarchical and, therefore, a partial order, it is reasonable to formulate corresponding requirements for the *under* relation.

$MP_1 = Graph +$
prds
$under\colon N\ N$ (subnode of)
axms
inheritance

$$x \in N : under(x, x) \quad \text{(reflexivity)}$$

$$x, y \in N : under(x, y) \wedge under(y, x) \Rightarrow x = y \quad \text{(antisymmetry)}$$

$$x, y, z \in N : under(x, y) \wedge under(y, z) \Rightarrow under(x, z) \quad \text{(transitivity)}$$

Fig. 6: The MP_1 specification including the predicate *under*

While the use of the predicate *under* is quite natural at the schema level, the question arises how it is to be interpreted at the instance level. Typically, objects are seen as monolithic entities even if they are mapped to multiple types in the class hierarchy. In this paper, we follow a different approach and consider objects to consist of a set of interconnected parts called *particles*. Each particle is represented by a node and is typed in a specific class in the schema. The advantage of this approach is that the structure of an object is made visible and

[7] The "MP" stands for "Model Part" and describes the fact that systems for this model represent only a part (schema or instance) of the whole object-oriented system.

resembles the object's type hierarchy at the schema level allowing proper typing of links.[8] Figure 7 shows an exemplary instance level for the schema in Fig. 3b.[9]

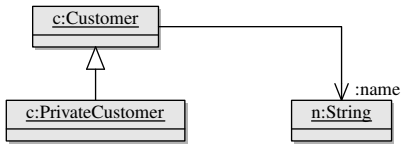


Fig. 7: Objects represented by particles

The model currently allows an object to contain more than one particle for the same type. This is typically forbidden by object-oriented languages.[10] In order to implement this requirement, we want to specify something like that:

$$(1) \qquad x, y \in N : \mathit{rel}(x, y) \wedge \mathit{type}(x) = \mathit{type}(y) \Rightarrow x = y \qquad \text{(unique particles)}$$

Here, another predicate *rel* has been used which shall be fulfilled when two particles belong to the same object and, therefore, are *rel*ated. Obviously, this predicate describes an equivalence relation as it is reflexive, symmetric, and transitive. Furthermore, the equivalence comprises the *under* relation, because each two particles connected by the *under* relation belong to the same object and, consequently, are part of the *rel* relation.[11] The resulting specification MP_2 is shown in Fig. 8.[12]

Another issue currently not resolved is association multiplicity: In our model, all associations are many-to-many, because the number of links at the instance level is not restricted in any way. However, many-to-one associations are often necessary in object-oriented schemas to allow at most one linked target object for any given association and source object. To achieve this, a formula like the following one is necessary which disallows the existence of two links which are instances of the same association and start at the same particle:[13]

[8] For the purpose of typing, simple homomorphisms are sufficient; there is no need to introduce homomorphisms "up to inheritance".

[9] We do not use a different notation for schema inheritance and the relationship between particles because it can be easily deduced from the context which relation is meant.

[10] An exception to this rule is the programming language *C++* which explicitly allows this behaviour [14].

[11] Note, however, that the *rel* relation might not be *generated* by the *under* relation. That means that there may exist related particles that do *not* belong to the same object. However, this is avoided in all practical examples.

[12] Note that reflexivity of the *rel* relation need not be specified by an axiom as it is a consequence from the combination of the first inheritance axiom and the last component axiom.

[13] Note that this axiom disallows multi-valued associations completely. This is desired, however, as only single-valued associations can be dereferenced at the instance level

$MP_2 = MP_1 +$
prds
$rel\colon N\ N$ (related to)
axms
components

$$x, y \in N : rel(x, y) \Rightarrow rel(y, x) \quad \text{(symmetry)}$$

$$x, y, z \in N : rel(x, y) \wedge rel(y, z) \Rightarrow rel(x, z) \quad \text{(transitivity)}$$

$$x, y \in N : under(x, y) \Rightarrow rel(x, y) \quad \text{(components)}$$

Fig. 8: The MP_2 specification including the predicate *rel*

(2)

$$x, y \in E : source(x) = source(y) \wedge type(x) = type(y) \Rightarrow x = y \quad \text{(at most one)}$$

The axioms (1) and (2) are called *typing axioms*.

The last issue is the integration of software constructs, namely operations, parameters, messages, arguments, and methods. In order to model operations and messages, the specification MP_2 is extended by a unary predicate called *software* which distinguishes between class nodes and operation nodes in schemas and between object nodes and message nodes in instances. The distinction between association edges and parameter edges on the one hand and between link edges and argument edges on the other hand is deduced from the context: If an edge starts at a class/object it is considered an association/link, otherwise it constitutes a parameter/argument. The resulting specification is shown in Fig. 9.[14]

An example of an operation is displayed in Fig. 10a, a message for this operation is shown in Fig. 10b. The modelling of methods builds upon the mapping of messages and arguments into the model and is described in the next section.

We use the following notation: $\mathbf{Alg}(MP)$ denotes the category of all MP-systems and MP-homomorphisms. The arrow category $\mathbf{Alg}(MP)^2$ consists of all typed instances which do not necessarily fulfil the typing axioms. The full subcategory $\mathbf{Sys} \subseteq \mathbf{Alg}(MP)^2$ restricts the arrow category to those typed instances conforming to these axioms. Given a fixed schema system S, the slice category $\mathbf{Alg}(MP)\downarrow S$ expresses the category of all typed instances for the system S, and the category $\mathbf{Sys}(S)$ denotes the full subcategory of $\mathbf{Alg}(MP)\downarrow S$ whose objects fulfil the typing axioms.[15]

---

in a well-defined way. Multi-valued associations need further information (e. g. an index) when accessing links, which does not fit well in our graph structure model.

[14] The model allows parameters to point to operations; this is reasonable as it enables to model basic statements like *if-then-else* as operations.

[15] Obviously, $\mathbf{Sys}(S)$ is also a subcategory of $\mathbf{Sys}$.

$MP =$

sorts

N	(nodes)
E	(edges)

opns

$s\colon E \to N$	(source node of an edge)
$t\colon E \to N$	(target node of an edge)

prds

$under\colon N\ N$	(subnode of)
$rel\colon N\ N$	(related to)
$software\colon N$	(software part vs. data part)

axms

inheritance

(3) $x \in N : under(x, x)$ (reflexivity)

(4) $x, y \in N : under(x, y) \wedge under(y, x) \Rightarrow x = y$ (antisymmetry)

(5) $x, y, z \in N : under(x, y) \wedge under(y, z) \Rightarrow under(x, z)$ (transitivity)

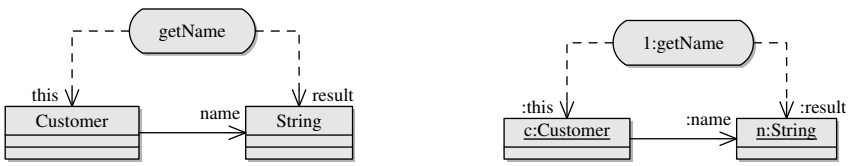
components

(6) $x, y \in N : rel(x, y) \Rightarrow rel(y, x)$ (symmetry)

(7) $x, y, z \in N : rel(x, y) \wedge rel(y, z) \Rightarrow rel(x, z)$ (transitivity)

(8) $x, y \in N : under(x, y) \Rightarrow rel(x, y)$ (components)

Fig. 9: The complete *MP* specification



(a) Operation with parameters (b) Message with arguments

Fig. 10: Software constructs

Furthermore, there exists a functor $\mathcal{F}\colon \mathbf{Alg}(MP)^2 \to \mathbf{Sys}$ which transforms any typed instance by factoring through the congruence generated by the axioms such that the resulting typed instance fulfils the typing axioms. This functor is an epireflector because of the freeness property of the factorisation. This functor never changes the schema:

Lemma 3.1 ([7, Lemma 13.13]). *Let $D ::= I \xrightarrow{type_I} S$ be a typed instance, and let $\mathcal{F}_{\mathrm{Ob}}(D)$ be the typed instance $I' \xrightarrow{type_{I'}} S'$. Then $S \cong S'$ holds.* □

From this lemma, it follows that the functor $\mathcal{F}$ can be restricted to a slice category for some fixed schema S, resulting in a family of epireflectors $\mathcal{F}^S\colon \mathbf{Alg}(MP)\downarrow S \to \mathbf{Sys}(S)$ for each possible schema S.

Summarising our results so far, an object-oriented schema is modelled as an *MP*-system S. An instance of this schema consists of an *MP*-system I and a typing *MP*-homomorphism $type\colon I \to S$ such that $I \xrightarrow{type} S$ is an object of the category $\mathbf{Sys}(S)$. Every schema instance $type\colon I \to S$ in $\mathbf{Alg}(MP)\downarrow S$ can be uniquely transformed into an object of the category $\mathbf{Sys}(S)$ by the epireflector $\mathcal{F}^S$.

4 Methods

A *method* is part of a program and specifies how the program reacts on a message for a certain operation. It constitutes an *implementation* of an operation. Here, the set of operations consists not only of "user-defined" operations but also of operations for evaluating expressions and for representing statements.[16] In other words, for each construct which influences the behaviour of a process, there exists a corresponding operation. A *program* is then a collection of methods such that all operations for which messages exist are implemented.

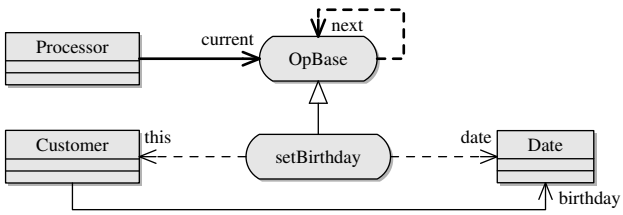
Each method is implemented by a single *DPO rule* [6] which is properly typed in the schema S. A typed DPO rule is a span $L \xleftarrow{l} K \xrightarrow{r} R$ together with the typings $L \xrightarrow{type_L} S$, $K \xrightarrow{type_K} S$, and $R \xrightarrow{type_R} S$, where L, K, R, and S are *Graph* systems and l and r are *Graph* homomorphisms such that $type_L \circ l = type_K = type_R \circ r$. The left part of the rule describes the required process state necessary for executing this method and contains at least a message typed in the operation this method implements. The remainder of the rule consists of the gluing part and the right part and specifies how this state is changed by method execution. Generally, the gluing part is the common subgraph of both the left and the right part of the rule.[17]

In order to be able to determine which message is ready to be processed, a special "marker" object called *processor* is used. A message referenced by a processor through a special *current* link is called *active*. Methods are formulated

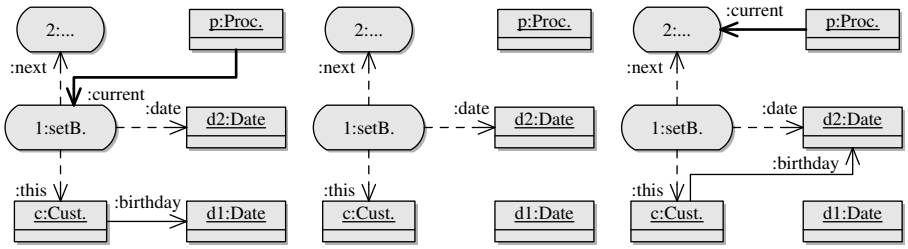
[16] For example, the addition of two integer values or the *if-then-else* statement are both represented by suitable operations.

[17] This means that both morphisms of the DPO rule are injective.

such that their left part requires an active message. Additionally, each method moves the processor object to the next message according to the flow of control. This next message is determined by a special argument called *next*.[18] Multiple processor objects can be used to model multi-threaded processing. Figure 11 displays the DPO rule for a method changing the target of a link.[19]



(a) Schema



(b) Left part (c) Gluing part (d) Right part

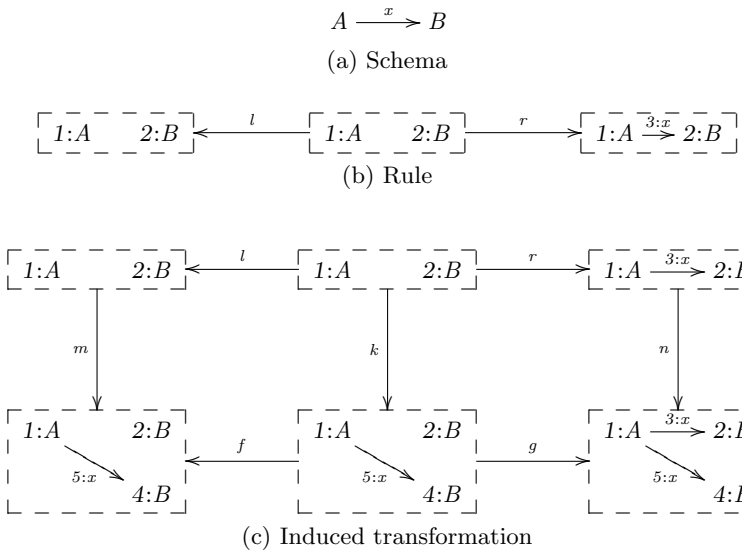
Fig. 11: Example method "Change target of *birthday* link"

A method is executed by applying the underlying DPO rule along a match to the graph structure describing the instance world, i. e., objects, links, messages, and arguments. According to the DPO model, in the first step a pushout complement has to be computed to complete the left side. In the second step, the right side is built by a pushout. However we cannot do this in our category $\mathbf{Sys}(S)$ as neither do pushout complements exist nor are pushouts along monomorphisms van-Kampen squares [6] in all cases. Therefore, we perform DPO transformations which are typed in a schema S in the slice category $\mathbf{Alg}(MP^*)\downarrow S$, where MP^* is the signature obtained by removing all axioms from MP, and provide sufficient conditions that guarantee the fulfilment of the axioms after transformation. These conditions are necessary as not all DPO transformations yield typed instances

[18] Only very few messages do not have a *next* argument. This includes the *end* message which terminates process execution and the *if-then-else* message which contains a *then* and a *else* argument instead.

[19] The example shows that operations and messages can also be specialised and possess a particle structure (the particle structure of the messages is not displayed for clarity). This is used to allow processor objects to point to any message.

which fulfil all the axioms. The following figures demonstrate two such counter examples: adding a link violates axiom (2) (Fig. 12), and eliminating inheritance violates axiom (5) (Fig. 13). In the figures, the element-wise mapping of the homomorphisms is indicated by equally named nodes and edges, and frames are used to group the elements belonging to a single graph.



(a) Schema

(b) Rule

(c) Induced transformation

Fig. 12: Adding a link violates axiom (2) on page 328

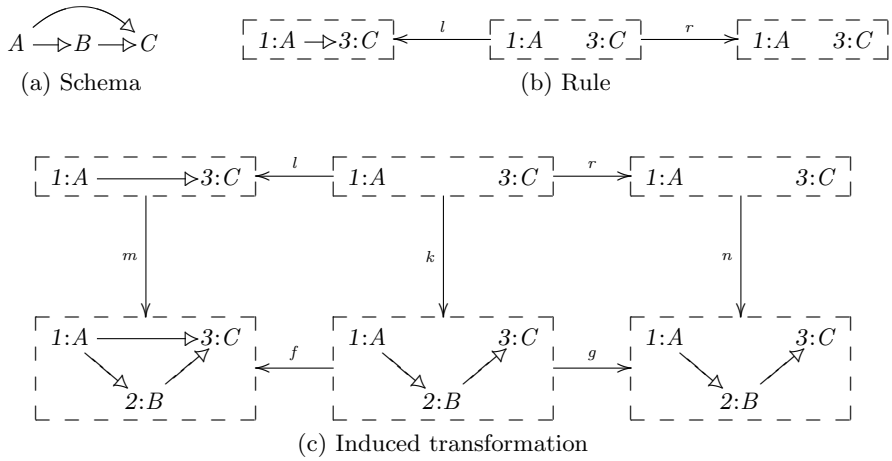
In order to rule out situations as depicted in Fig. 12 we need DPO rules that *pull back edges*. Such rules only add an edge at the right side if no other edge exists which starts at the same node.

Definition 4.1 (DPO rules pulling back edges). *Let $\Sigma = (S, OP, P)$ be an extended signature, and let $L \xleftarrow{l} K \xrightarrow{r} R$ be a DPO rule. Then the DPO rule* pulls back edges *if for each edge $e_R \in R_E$ there is a node $k \in K_N$ and an edge $e_L \in L_E$, such that the equations*

$$\begin{aligned} source^L(e_L) &= l_N(k) \\ source^R(e_R) &= r_N(k) \\ type_{L,E}(e_L) &= type_{R,E}(e_R) \end{aligned}$$

hold.

In order to rule out situations as depicted in Fig. 13, we restrict DPO rules to *completing* homomorphisms which "pull back" relations. Completing



(a) Schema (b) Rule

(c) Induced transformation

Fig. 13: Eliminating inheritance violates axiom (5) on page 329

homomorphisms are an extension to strictly full homomorphisms:[20] While a strictly full homomorphism h only "pulls back" a relation if all related elements are known to be in the range of h, a completing homomorphism h "pulls back" relations even if only a *part* of the related elements is known to be reached.

Definition 4.2 (Completing homomorphism). *Let $\Sigma = (S, OP, P)$ be an extended signature, let $h\colon A \to B$ be a Σ-homomorphism between the Σ-systems A and B, and let $p \in P_w$ be a predicate over a sort word $w \in S^*$. Then h is* completing *on p if for every non-empty sort word w' resulting from eliminating arbitrary sorts from w and for each two tuples $x \in B_w$ and $x' \in A_{w'}$ the implication*

$$h_{w'}(x') = \langle x \rangle_{w'} \wedge x \in p^B \Rightarrow \exists y \in A_w : \langle y \rangle_{w'} = x' \wedge h_w(y) = x \wedge y \in p^A$$

holds, where the notation $\langle x \rangle_{w'}$ stands for the projection of the tuple x onto the elements of the sorts in w'. h is completing *if h is completing on all predicates.*

Now we are able to define *valid* DPO rules:

Definition 4.3 (Valid rule). *A DPO rule $L \xleftarrow{l} K \xrightarrow{r} R$ is* valid *iff it pulls back edges and l and r are completing homomorphisms.*

These restrictions do not have much impact on the expressiveness of methods. The first restriction requiring completing homomorphisms disallows changing the inner structure of objects by adding or removing particles. However, this is an unusual way of dealing with objects at runtime at best. The second restriction

[20] A homomorphism $h\colon A \to B$ is *strictly full* if $h_w(x) \in p^B \Rightarrow x \in p^A$ for all $x \in A_w$ and all predicates $p \in P_w$.

allows to add a link on the right side of a rule only if a similar link has previously been removed on the left side of the same rule. This is unproblematic if it can be ensured that there always exists a link for each (object, association) pair, which, for example, can initially point to a "null" object to indicate an uninitialised link.[21]

Now we can state the main theorem of this section:

Theorem 4.4 (Transformation preserves axioms [7, Theorem 14.29]). *Let S be an MP-system. Let $L \xleftarrow{l} K \xrightarrow{r} R$ be a valid rule in* $\mathbf{Sys}(S)$*, G a* $\mathbf{Sys}(S)$*-object, and $m\colon L \to G$ a match in* $\mathbf{Sys}(S)$*, such that the rule is applicable according to the DPO model. Let $G \xleftarrow{f} D \xrightarrow{g} H$ be the resulting transformation after applying the rule in* $\mathbf{Alg}(MP^*)\downarrow S$*. Then D and H fulfil all axioms and are, therefore,* $\mathbf{Sys}(S)$*-objects.* □

5 Model Transformation and Data Migration

So far we can describe object-oriented systems, consisting of typed data, programs, and processes. In this section we introduce schema transformations that can be uniquely extended to migrations of corresponding data and proceseses (the migration of programs is handled in the next section).[22]

Definition 5.1 (Transformation, Refactoring). *A transformation $t\colon S \overset{S^*}{\rightsquigarrow} S'$ in the category* $\mathbf{Alg}(MP)$ *is a* span $S \xleftarrow{l^t} S^* \xrightarrow{r^t} S'$*. Such a transformation is called a* refactoring *iff l^t is surjective.*

A general transformation allows reduction and unfolding as well as extension and folding through the use of non-surjective homomorphisms (reduction and extension) and non-injective homomorphisms (unfolding and folding) on the left and right side of the span, respectively. Refactorings are special transformations which are constrained to surjective homomorphisms on the left side of the span. This constraint comes from the fact that refactorings are not allowed to delete schema objects because such a deletion almost always causes information (data, programs and/or processes) at the instance level to be lost, which does not meet the intuitive requirement that a refactoring preserve information. In the following we use the term *schema transformation* if the span consists of schema objects, and *migration* if the span consists of typed instances.

Given a typed instance $I \xrightarrow{type_I} S$ and a schema transformation $t\colon S \overset{S^*}{\rightsquigarrow} S'$, the migration is performed as follows:

(1) $\mathcal{P}^{l^t}$, the pullback functor along l^t, is applied to $I \xrightarrow{type_I} S$, resulting in the typed instance $I^* \xrightarrow{type_{I^*}} S^*$. This part of the transformation is responsible

[21] [7] shows in full detail how this can be done.

[22] See [15–17] for precursor material on data migration induced by schema transformations.

for unfolding instance elements if l^t is not injective, and for deleting elements if l^t is not surjective.

(2) $\mathcal{F}^{r^t}$, the composition functor along r^t, is applied to $I^* \xrightarrow{type_{I^*}} S^*$, resulting in the typed instance $I^* \xrightarrow{r^t \circ type_{I^*}} S'$. This part of the transformation is used to retype instance elements and to add new types without any instances.

(3) $I^* \xrightarrow{r^t \circ type_{I^*}} S'$ may violate the typing axioms. Therefore, the epireflector $\mathcal{F}^{S'}$ into the subcategory $\mathbf{Sys}(S')$ is applied to it, resulting in the typed instance $I' \xrightarrow{type_{I'}} S'$ (the schema is left unchanged due to Lemma 3.1). This part of the transformation is responsible for identifying instance elements due to retyping.

These three steps are visualised in Fig. 14.

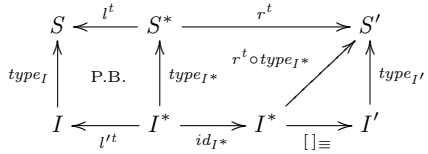


Fig. 14: Schema transformation and instance migration

The composition of the three functors results in the migration functor defined below:

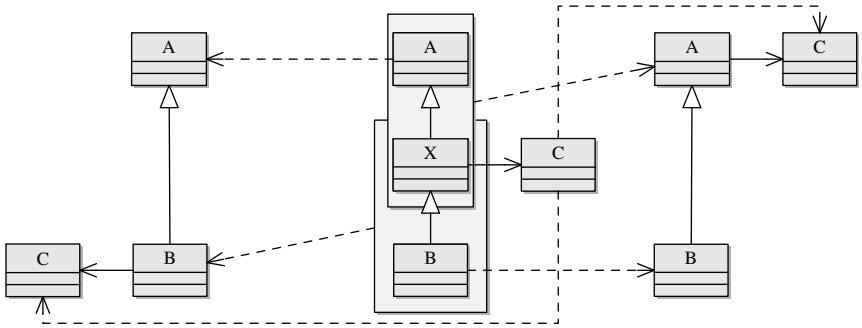
Definition 5.2 (Migration functor). *Let* $t\colon S \overset{S^*}{\rightsquigarrow} S'$ *be a transformation. The* migration functor $\mathcal{M}^t\colon \mathbf{Sys}(S) \to \mathbf{Sys}(S')$ *is then defined as:*

$$\mathcal{M}^t ::= \mathcal{F}^{S'} \circ \mathcal{F}^{r^t} \circ \mathcal{P}^{l^t} \ ,$$

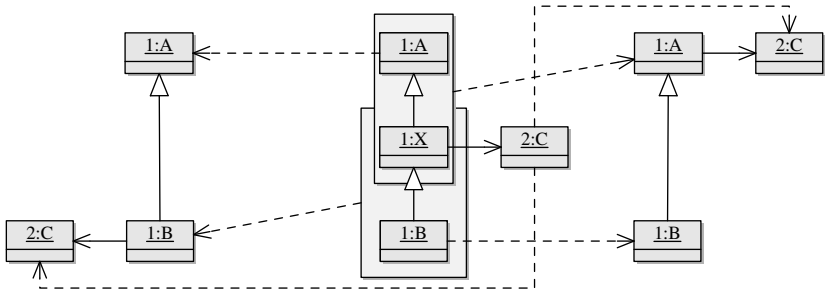
where the functor $\mathcal{P}^{l^t}\colon \mathbf{Alg}(MP)\downarrow S \to \mathbf{Alg}(MP)\downarrow S^*$ *is the pullback functor along* l^t*, the functor* $\mathcal{F}^{r^t}\colon \mathbf{Alg}(MP)\downarrow S^* \to \mathbf{Alg}(MP)\downarrow S'$ *is the composition functor along* r^t*, and the functor* $\mathcal{F}^{S'}\colon \mathbf{Alg}(MP)\downarrow S' \to \mathbf{Sys}(S')$ *is the epireflector into the subcategory* $\mathbf{Sys}(S')$*.*

Note that due to Lemma 3.1, the migration functor results in an instance that is correctly typed in the target schema S'.

The example in Fig. 15 shows a transformation which moves the origin of an association one level upwards the inheritance hierarchy and the induced migration of an exemplary instance. On the left side the class B is unfolded, yielding the two classes B and X in the middle, and the origin of the association is moved to the temporary class X. On the right side the class X is folded with the class A, such that the association starts at the class A after the transformation. The modification of objects and links by the induced migration is performed analogously. Note that the unfolding on the left is due to the pullback construction, and the folding on the right side is due to the epireflector which takes care that axiom (1) is satisfied.



(a) Schema transformation



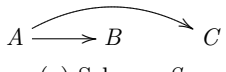
(b) Induced migration

Fig. 15: Moving the origin of an association upwards the inheritance hierachy

6 Method Migration

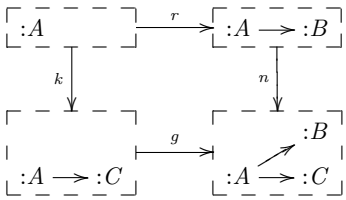
The migration of methods is performed in the same way as the migration of data and processes. But as methods are valid DPO rules according to Def. 4.3, it has to be ensured that their properties are preserved by a migration. Additionally, methods already executed which are represented by two pushout diagrams shall be transformed so that the resulting diagrams are again pushouts. This ensures that processes that have already been executed are compatible to the new schema after migration. However, this does not hold for arbitrary transformations. In Fig. 16, two classes B and C of a schema S are merged, resulting in the class BC in the schema S'. At the instance level, the right pushout of a method adding a link is presented. The migrated diagram is a pushout in the subcategory $\mathbf{Sys}(S')$ of all typed instances conforming to the typing axioms, but not a pushout in the category $\mathbf{Alg}(MP^*)\downarrow S'$ in which the migration is computed. This can be deduced from the elemental properties of pushouts (see e. g. [6]).

In order to migrate DPO rules and DPO diagrams properly we need to restrict the allowed transformations. We can show that if transformations are disallowed to fold associations on the right side, DPO rules can be migrated correctly in all cases. This results in the following definition of a proper transformation:

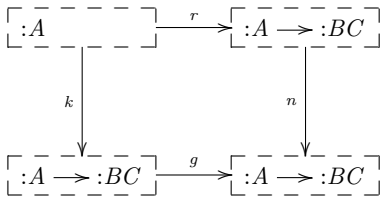


(a) Schema S

(b) Schema S'



(c) Right side of an applied DPO rule before the migration



(d) Right side of an applied DPO rule after the migration

Fig. 16: Pushout in $\mathbf{Alg}(MP^*)\downarrow S$ is not preserved by a migration

Definition 6.1 (Proper transformation). *A transformation* $S \xleftarrow{l^t} S^* \xrightarrow{r^t} S'$ *in* $\mathbf{Alg}(MP)$ *is* proper *if* r^t *is injective on associations, i. e., if* $r^t_E(x) = r^t_E(y) \Rightarrow x = y$ *holds for all* $x, y \in S^*_E$.

The correct migration of valid DPO rules is guaranteed by the following proposition:

Proposition 6.2 (Migration preserves valid DPO rules [7, Proposition 15.23]). *Given a proper transformation* $t\colon S \overset{S^*}{\rightsquigarrow} S'$, *let*

$$(I^1 \xrightarrow{type_{I^1}} S) \xleftarrow{l} (I^2 \xrightarrow{type_{I^2}} S) \xrightarrow{r} (I^3 \xrightarrow{type_{I^3}} S)$$

be a valid DPO rule. Then

$$\mathcal{M}^t(I^1 \xrightarrow{type_{I^1}} S) \xleftarrow{\mathcal{M}^t(l)} \mathcal{M}^t(I^2 \xrightarrow{type_{I^2}} S) \xrightarrow{\mathcal{M}^t(r)} \mathcal{M}^t(I^3 \xrightarrow{type_{I^3}} S)$$

is a valid DPO rule as well. □

The migration of DPO diagrams is ensured by the following proposition:

Proposition 6.3 (Migration preserves pushouts [7, Proposition 15.35]). *Let* $t\colon S \overset{S^*}{\rightsquigarrow} S' \mathrel{\widehat{=}} S \xleftarrow{l^t} S^* \xrightarrow{r^t} S'$ *be a proper transformation and* $(L \xrightarrow{type_L} S) \xleftarrow{l} (K \xrightarrow{type_K} S) \xrightarrow{r} (R \xrightarrow{type_R} S)$ *be a valid DPO rule. Let*

$$(D \xrightarrow{type_D} S) \xrightarrow{g} (H \xrightarrow{type_H} S) \xleftarrow{n} (R \xrightarrow{type_R} S)$$

be a pushout of

$$(D \xrightarrow{type_D} S) \xleftarrow{k} (K \xrightarrow{type_K} S) \xrightarrow{r} (R \xrightarrow{type_R} S)$$

in $\mathbf{Alg}(MP_*)\downarrow S$*, where all typed instances are in* $\mathbf{Sys}(S)$*. Then*

$$\mathcal{M}^t(D \xrightarrow{type_D} S) \xrightarrow{\mathcal{M}^t(g)} \mathcal{M}^t(H \xrightarrow{type_H} S) \xleftarrow{\mathcal{M}^t(n)} \mathcal{M}^t(R \xrightarrow{type_R} S)$$

is a pushout of

$$\mathcal{M}^t(D \xrightarrow{type_D} S) \xleftarrow{\mathcal{M}^t(k)} \mathcal{M}^t(K \xrightarrow{type_K} S) \xrightarrow{\mathcal{M}^t(r)} \mathcal{M}^t(R \xrightarrow{type_R} S)$$

in $\mathbf{Alg}(MP_*)\downarrow S'$*, where all typed instances are in* $\mathbf{Sys}(S')$*.* □

Both propositions can be combined, yielding the following theorem:

Theorem 6.4 (Correctness of the migration of programs [7, Theorem 15.36]). *Let* $t\colon S \overset{S^*}{\rightsquigarrow} S'$ *be a proper transformation. Then the migration functor* $\mathcal{M}^t$ *transfers the validity of non-applied methods (DPO rules) and applied methods (DPO transformations) from the category* $\mathbf{Alg}(MP)\downarrow S$ *into the category* $\mathbf{Alg}(MP)\downarrow S'$*.*

Proof. Direct consequence of Proposition 6.2 and Proposition 6.3. □

7 Outlook

With the framework presented above, a major step towards migration of complete object-oriented systems is proposed. Certainly, the framework is not universal as it is subject to some (reasonable) constraints. Migrations are considered to be instances of transformations. The innovative part of the theory described consists of the *automatic* transformation of a migration source, computing the target with the help of a functor on slice categories. This functor is composed of three factors: Generally, the pullback functor $\mathcal{P}^{l^t}$ is right-adjoint where the second factor—the composition functor $\mathcal{F}^{r^t}$—is its left-adjoint. But the third factor—the construction $\mathcal{F}^S$ into the subcategory $\mathbf{Sys}(S)$—yields an adjunction as well. Thus, the whole migration enjoys well-understood universal properties which can further be pursued into three different directions.

The first direction for future research will be the development of tools that support migration induced by refactoring rules. If transformation rules can be captured ergonomically in an appropriate application, migrations can automatically and *uniquely* (by adjointness) be computed. Thus, content migration of databases is possible as well as migration of running processes in a software system. These tools should discover potential for composition, as well: Bigger refactorings should be decomposable into elementary changes, atomic steps must be proved to combine to more comprehensive procedures. This is another facet for future research.

Theorem 6.4 states that dynamical semantics is preserved by refactorings where semantics is based on valid DPO rules. Hence the second direction is to find a comparable correctness criterion for data. This must include a formal specification of "information" to distinguish between semantics-preserving refactorings and information-distorting transformations.

The third direction consists of abstracting away from pure graph structures. It has to be investigated to what extent the results can be generalised to elementary topoi or even to adhesive categories [6, 18]. An approach can be found in [15] which covers data migration only. Hence, an extension to method migration is desirable.

References

[1] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
[2] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2002)
[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional (1995)
[4] Fowler, M., Scott, K.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley (2003)
[5] Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions On Software Engineering **30**(2) (2004) 126–139
[6] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
[7] Schulz, C.: Refactoring objektorientierter Systeme. Forschungsberichte der FHDW Hannover **2** (2009)
[8] Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: ICGT. (2004) 383–398
[9] Kastenberg, H., Kleppe, A.G., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In Gorrieri, R., Wehrheim, H., eds.: Proceedings of the 8th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems, Bologna, Italy. Volume 4037 of Lecture Notes in Computer Science., London, Springer Verlag (June 2006) 186–201
[10] Kastenberg, H., Kleppe, A.G., Rensink, A.: Engineering object-oriented semantics using graph transformations. Technical Report CTIT Technical Report 06-12, University of Twente (2006)
[11] Bardohl, R., Ehrig, H., de Lara, J., Runge, O., Taentzer, G., Weinhold, I.: Node type inheritance concept for typed graph transformation. Technical Report Technical Report 2003-19, Technical University, Berlin (2003)
[12] de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. Theoretical Computer Science **376**(3) (2007) 139–163
[13] Mal'cev, A.I.: Algebraic systems. Springer (1973)
[14] International Organization for Standardization: ISO/IEC 14882:2003: Programming languages – C++, Genf, Schweiz. (2003)
[15] König, H., Löwe, M., Schulz, C.: Functor semantics for refactoring-induced data migration. Forschungsberichte der FHDW Hannover **1** (2007)
[16] Löwe, M., König, H., Schulz, C., Peters, M.: Refactoring information systems – a formal framework. In: Proceedings WMSCI 2006. Volume 1. (2006) 75–80

[17] Löwe, M., König, H., Schulz, C., Peters, M.: Refactoring information systems – handling partial composition. In: Electronic Communications of the EASST. Volume 3. (2006)
[18] Goldblatt, R.: Topoi: The Categorical Analysis of Logic. Dover Publications (1984)

Christoph Schulz

Fachhochschule für die Wirtschaft Hannover
Freundallee 15
D-30173 Hannover (Germany)
christoph.schulz@fhdw.de

Prof. Dr. Michael Löwe

Fachhochschule für die Wirtschaft Hannover
Freundallee 15
D-30173 Hannover (Germany)
michael.loewe@fhdw.de

Hans-Jörg Kreowski supervised Michael Löwe's diploma thesis at TU Berlin in 1981, and was the external examiner of his doctoral thesis in 1990.

Prof. Dr. Harald König

Fachhochschule für die Wirtschaft Hannover
Freundallee 15
D-30173 Hannover (Germany)
harald.koenig@fhdw.de