Electronic Communications of the EASST
Volume 26 (2010)

Manipulation of Graphs, Algebras and Pictures

Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday

Categorical Framework for the Transformation of Object-Oriented Systems: Operations and Methods

Christoph Schulz, Michael Löwe, and Harald König

21 pages

Guest Editors: Frank Drewes, Annegret Habel, Berthold Hoffmann, Detlef Plump
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Categorical Framework for the Transformation of Object-Oriented Systems: Operations and Methods

Christoph Schulz[1], **Michael Löwe**[1], **and Harald König**[1]

[1] Fachhochschule für die Wirtschaft (FHDW), Hannover
Freundallee 15, 30173 Hannover, Deutschland

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 extends the model described in [SLK10] by operations, messages, and methods, which allows to model *complete* object-oriented systems.[1] 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 categorical 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.

Keywords: Refactoring, Evolution, Transformation, Migration, Software

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" [Fow99]. 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 [GHJV95, Fow02]. 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,

[1] We call an object-oriented system *complete* if it consists of data, programs, and processes.

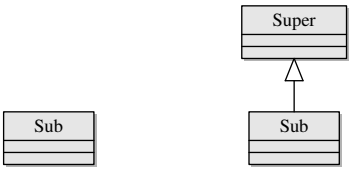


Figure 1: Refactoring “Introduce a new superclass”

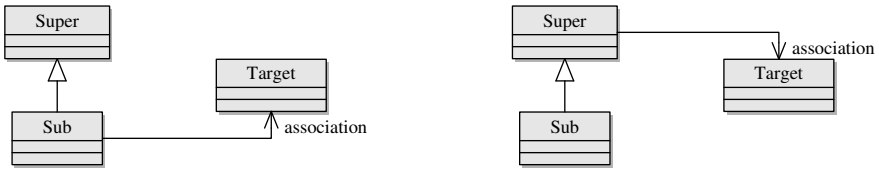


Figure 2: Refactoring “Move the origin of an association from a subclass to a superclass”

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*.

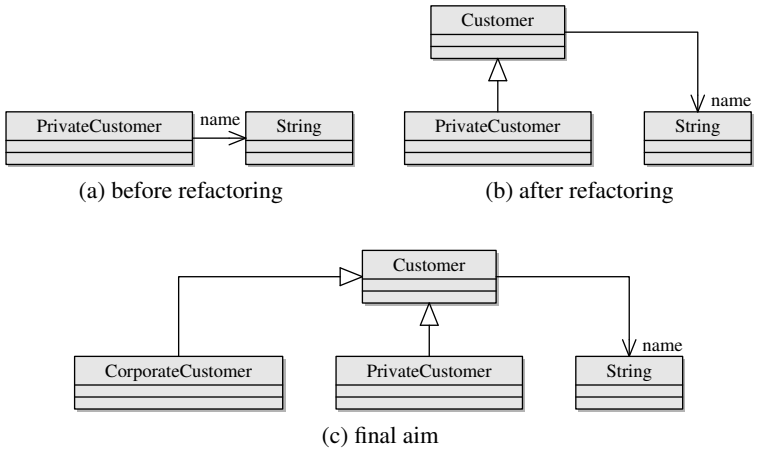
Two typical object-oriented refactorings are “Introduce a new superclass” (Fig. 1) and “Move the origin of an association from a subclass to a superclass”, as shown in Fig. 2. A combined application of these two refactorings on the schema in Fig. 3a could be used to prepare the model for an extension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

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 about refactoring and induced migration of *whole systems*, which we define to consist of data, programs, and processes, all typed in the same schema.[3]

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)

[2] All class diagrams are specified in the UML [FS03].

[3] See [MT04] and especially the bibliography contained therein for a general overview on software refactoring.



(a) before refactoring (b) after refactoring

(c) final aim

Figure 3: Refactoring an exemplary object-oriented model

refactorings, and *migrations*. This data model is described in more detail in [SLK10].

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 the use of *graph structure transformations*. This paper chooses the DPO approach for describing and applying graph structure transformations[4]. 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 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

[4] DPO stands for "Double Pushout"; the approach is presented in e. g. [EEPT06].

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.

2 Related Work

There exist approaches for modelling programs as algebraic graph transformation rules [CDFR04, KKR06a, KKR06b].[5] However, they fail in various ways to be suitable for our purposes. The approach in [CDFR04] 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 [KKR06a, KKR06b] 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 [BEL+03, EEPT06, LBE+07]. 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.

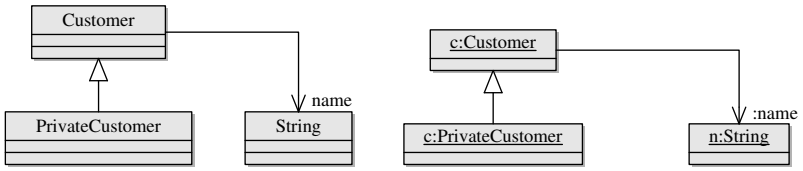
3 Models and Instances

The schema and the instance level of object-oriented systems are modelled by systems wrt. an extended specification.[6] 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, called *axioms*. 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* $(op^A : A_w \to A_s)_{w \in S^*, s \in S, op \in OP_{w,s}}$, 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.[7] A *system* A wrt. an extended specification $Spec = (\Sigma, H(X))$ is a Σ-system such that all axioms in $H(X)$ are valid in A. A Σ-*homomorphism* $h : A \to B$ between two Σ-systems A and B wrt. an extended signature $\Sigma = (S, OP, P)$ is a family of mappings $(h_s : A_s \to B_s)_{s \in S}$, 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 $op : w \to s$ and $h_w(p^A) \subseteq p^B$ for all predicates $p : w$ where $w = s_1 s_2 \ldots s_n \in$

[5] [KKR06b] is the technical report [KKR06a] is based on and goes into much more detail about the presented programming language *TAAL*.

[6] See [Mal73] for the special case when signatures consist of one sort only.

[7] Given $w = s_1 s_2 \ldots s_n$, A_w is a short-hand notation for the product set $A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$.



(a) Schema (b) Instance

Figure 4: Example of a schema together with a typed instance

S^*.[8] Each Σ-homomorphism $h\colon A \to B$ between two *Spec*-systems A and B wrt. an extended specification $Spec = (\Sigma, H(X))$ is called a *Spec*-homomorphism.

In the following we shortly summarise the underlying model for classes and associated data as described in [SLK10]. We use a graph-like signature enriched by axioms to describe both schemas and schema instances. Classes are represented as graph nodes of the sort N. Associations and links are represented as graph edges of the sort E. Class inheritance is modelled by a binary predicate *under*: If, in a system S, a 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).[9] The binary predicate *rel* represents the equivalence closure of *under*. On the instance level, the predicates *under* and *rel* are used to model objects as collections of related *particles*. Each particle is represented by a node of the sort N 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 resembles the object's type hierarchy at the schema level allowing proper typing of links.[10] Figure 4 shows an exemplary schema together with one possible typed instance.[11]

This model is capable of representing object-oriented data typed in a schema. However, we need software constructs, namely operations, parameters, messages, arguments, and methods, as well. In order to model operations and messages, the specification 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.[12] This yields the following specification:

$MP =$
sorts
N (nodes)
E (edges)

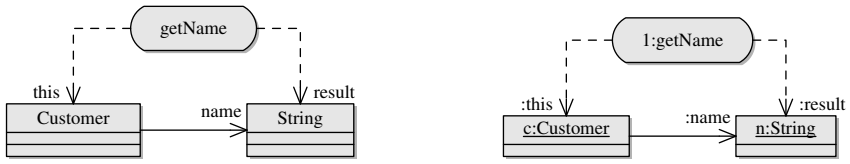
[8] Given $w = s_1 s_2 \ldots s_n$, $h_w(x_1, x_2, \ldots, x_n)$ is a short-hand notation for the tuple $(h_{s_1}(x_1), h_{s_2}(x_2), \ldots, h_{s_n}(x_n))$.

[9] It follows from the properties of subclassing that the predicate *under* is a partial order.

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

[11] Note that the instance graphs do not constitute proper UML diagrams: Instead we use the *schema* inheritance notion to depict explicit object decompositions into particles.

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



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

Figure 5: Software constructs

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

$$x \in N : under(x,x) \quad \text{(reflexivity)} \quad \text{(MP.1)}$$

$$x,y \in N : under(x,y) \wedge under(y,x) \Rightarrow x = y \quad \text{(antisymmetry)} \quad \text{(MP.2)}$$

$$x,y,z \in N : under(x,y) \wedge under(y,z) \Rightarrow under(x,z) \quad \text{(transitivity)} \quad \text{(MP.3)}$$

components

$$x,y \in N : rel(x,y) \Rightarrow rel(y,x) \quad \text{(symmetry)} \quad \text{(MP.4)}$$

$$x,y,z \in N : rel(x,y) \wedge rel(y,z) \Rightarrow rel(x,z) \quad \text{(transitivity)} \quad \text{(MP.5)}$$

$$x,y \in N : under(x,y) \Rightarrow rel(x,y) \quad \text{(components)} \quad \text{(MP.6)}$$

An example of an operation is displayed in Fig. 5a, a message for this operation is shown in Fig. 5b. The modelling of methods builds upon the mapping of messages and arguments into the model and is described in the next section.

In [SLK10], the model is further extended by two constraints. First, for each object there can only exist at most one particle of a given type. Second, associations are many-to-one, such that we are able to deterministically access links by methods (see next section). As these constraints depend on both the instance *and* the schema, we have to combine these two parts in order to be able to formulate the constraints as implications. For this, we internalise the typing homomorphism *type*:

Definition 1 (Specification M) The specification M consists of

- *two* copies of *MP*, one for the *schema part* where the sorts, operation symbols, and predicates are suffixed by "S", and one for the *instance part*, where the sorts, operation symbols, and predicates are suffixed by "I";

- two operation symbols *typeN* and *typeE* representing the typing;
- the *typing axioms* modelling unique particles within objects and many-to-one associations; and
- the *homomorphism axioms* ensuring that $(typeN, typeE)$ behave like a homomorphism in every M-system.

$M =$

sorts

NS	(schema nodes)
NI	(instance nodes)
ES	(schema edges)
EI	(instance edges)

opns

$typeN : NI \to NS$	(node typing)
$typeE : EI \to ES$	(edge typing)
$sS : ES \to NS$	(source node of a schema edge)
$sI : EI \to NI$	(source node of an instance edge)
$tS : ES \to NS$	(target node of a schema edge)
$tI : EI \to NI$	(target node of an instance edge)

prds

$underS : NS\ NS$	(subclass of)
$underI : NI\ NI$	(subparticle of)
$relS : NS\ NS$	(in class hierarchy of)
$rel : NI\ NI$	(in object of)
$softwareS : NS$	(class vs. operation)
$softwareI : NI$	(object vs. message)

axms

homomorphism axioms

$$x \in EI : typeN(sI(x)) = sS(typeE(x)) \quad \text{(M.1)}$$

$$x \in EI : typeN(tI(x)) = tS(typeE(x)) \quad \text{(M.2)}$$

$$x, y \in NI : underI(x, y) \Rightarrow underS(typeN(x), typeN(y)) \quad \text{(M.3)}$$

$$x, y \in NI : relI(x, y) \Rightarrow relS(typeN(x), typeN(y)) \quad \text{(M.4)}$$

$$x \in NI : softwareI(x) \Rightarrow softwareS(typeN(x)) \quad \text{(M.5)}$$

inheritance axioms

$$x \in NS : underS(x, x) \quad \text{(reflexivity)} \quad \text{(M.6)}$$

$$x, y \in NS : underS(x, y) \wedge underS(y, x) \Rightarrow x = y \quad \text{(antisymmetry)} \quad \text{(M.7)}$$

$$x,y,z \in NS : underS(x,y) \wedge underS(y,z) \Rightarrow underS(x,z) \quad \text{(transitivity)} \quad \text{(M.8)}$$

$$x \in NI : underI(x,x) \quad \text{(reflexivity)} \quad \text{(M.9)}$$

$$x,y \in NI : underI(x,y) \wedge underI(y,x) \Rightarrow x = y \quad \text{(antisymmetry)} \quad \text{(M.10)}$$

$$x,y,z \in NI : underI(x,y) \wedge underI(y,z) \Rightarrow under(x,z) \quad \text{(transitivity)} \quad \text{(M.11)}$$

component axioms

$$x,y \in NS : relS(x,y) \Rightarrow relS(y,x) \quad \text{(symmetry)} \quad \text{(M.12)}$$

$$x,y,z \in NS : relS(x,y) \wedge relS(y,z) \Rightarrow rel(x,z) \quad \text{(transitivity)} \quad \text{(M.13)}$$

$$x,y \in NS : underS(x,y) \Rightarrow relS(x,y) \quad \text{(components)} \quad \text{(M.14)}$$

$$x,y \in NI : relI(x,y) \Rightarrow relI(y,x) \quad \text{(symmetry)} \quad \text{(M.15)}$$

$$x,y,z \in NI : relI(x,y) \wedge relI(y,z) \Rightarrow rel(x,z) \quad \text{(transitivity)} \quad \text{(M.16)}$$

$$x,y \in NI : underI(x,y) \Rightarrow relI(x,y) \quad \text{(components)} \quad \text{(M.17)}$$

typing axioms

$$x,y \in NI : rel(x,y) \wedge typeN(x) = typeN(y) \Rightarrow x = y \quad \text{(unique particles)} \quad \text{(M.18)}$$

$$x,y \in EI : sI(x) = sI(y) \wedge typeE(x) = typeE(y) \Rightarrow x = y \quad \text{(at most one target)} \quad \text{(M.19)}$$

We use the following notation: $\mathbf{Alg}(MP)$ denotes the category of all MP-systems and MP-homomorphisms; equivalently, $\mathbf{Alg}(M)$ denotes the category of all M-systems and M-homomorphisms. The objects of the arrow category $\mathbf{Alg}(MP)^2$ are $\mathbf{Alg}(MP)$-arrows $I \xrightarrow{type_I} S$, which do not necessarily fulfil the typing requirements (M.18) and (M.19).[13] The full subcategory $\mathbf{Sys} \subseteq \mathbf{Alg}(MP)^2$ restricts the arrow category to those arrows conforming to these requirements.[14] Given a fixed schema system S, the slice category $\mathbf{Alg}(MP)\downarrow S$ expresses the category of all $\mathbf{Alg}(MP)$-arrows into the system S, and the category $\mathbf{Sys}(S)$ denotes the full subcategory of $\mathbf{Alg}(MP)\downarrow S$ whose objects fulfil (M.18) and (M.19).[15]

We are now able to express properly typed instances in two ways, either as an M-system or as an $\mathbf{Alg}(MP)$-arrow in $\mathbf{Sys}$. Formally, these categories are isomorphic. In order to prove that, we first show that $\mathbf{Alg}(MP)^2$ is isomorphic to $\mathbf{Alg}(M')$ where M' is defined as below.

Definition 2 (Specification M') The specification M' is M without the typing axioms (M.18) and (M.19).

Lemma 1 $\mathbf{Alg}(M')$ *and* $\mathbf{Alg}(MP)^2$ *are isomorphic.*

[13] Let S be the MP-model $A \rightarrow\!\triangleright B \twoheadrightarrow C$. Then typing the MP-model $1{:}A \rightarrow\!\triangleright 2{:}B \triangleleft\!\leftarrow 3{:}A$ in S contradicts requirement (M.18), and typing the MP-model $2{:}C \twoheadleftarrow 1{:}B \twoheadrightarrow 3{:}C$ in S contradicts requirement (M.19).

[14] A subcategory $\mathbf{D} \subseteq \mathbf{C}$ is *full* if for each pair (A,B) of $\mathbf{D}$-objects, the sets of morphisms between A and B in $\mathbf{D}$ and $\mathbf{C}$ are identical.

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

Proof. See [SLK10, Lemma 4]. □

Lemma 2 **Alg**(M) *and* **Sys** *are isomorphic.*

Proof. This follows directly from Lemma 1. □

For Horn clause specifications $Spec = (\Sigma, H_\Sigma(X))$ where the signature only contains sorts and operation symbols, it is a well-known fact in universal algebra that the resulting category **Alg**$(Spec)$ is closed under the formation of products and extremal subobjects (see e. g. [Wec92, Theorem 14] for the single-sorted case).[16] This result has been extended to signatures including predicates in [Sch09a]. From [AHS04, Theorem 16.8], it follows that **Alg**(M) is a full and isomorphism-closed epireflective subcategory of **Alg**(M').[17] "Reflective" means that for each **Alg**(M')-object A, there exists an M'-morphism $u\colon A \to \mathscr{F}A$ into an **Alg**(M)-object $\mathscr{F}A$ which constitutes some sort of a "best approximation" of A in **Alg**(M), i. e., A is changed as little as possible in order to conform to the axioms in M. This is important as we do not want to change a system more than necessary in order to make it conformant to a set of axioms. (In fact, deleting all but one element from each set (and adjusting the mappings accordingly) always results in a conformant system with regard to *any* specification with positive Horn formulas, but is surely not desirable in our context.) The morphism u is called an **Alg**(M)*-reflection of* A and is an epimorphism if the subcategory is epireflective (as in our case). This reflection property can be extended to a functor, called *(epi)reflector* for (epi)reflective subcategories.[18] So we obtain the following proposition:

Proposition 1 (Epireflector $\mathscr{F}$) *There exists an epireflector* $\mathscr{F}\colon$ **Alg**$(MP)^2 \to$ **Sys**.

Proof. This follows immediately from Lemma 1 and Lemma 2. □

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 **Sys**(S). Every schema instance $type\colon I \to S$ in **Alg**$(MP)\!\downarrow\!S$ can uniquely be transformed into an object of the category **Sys** by the epireflector $\mathscr{F}$.

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

[16] Given a Horn clause specification $Spec$ and two models $A, B \in$ **Alg**$(Spec)$, A is an *extremal* subobject of B if it is a subobject of B and if every predicate over elements in the A-part of B that is true in B is also true in A. In other words, an extremal subobject is "as true as possible".

[17] A subcategory $\mathbf{D} \subseteq \mathbf{C}$ is *isomorphism-closed* if for each $A \in \mathrm{Ob}^{\mathbf{D}}$ and each $\mathbf{C}$-isomorphism $i\colon A \to A'$, it follows that $A' \in \mathrm{Ob}^{\mathbf{D}}$.

[18] See [AHS04, pp. 52] fore more information about reflective subcategories and reflectors.

representing statements.[19] 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* [EEPT06] 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 graphs and l and r are injective graph morphisms 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. Note that a method's DPO rule represents the change in program state when *calling* the method, which does *not* include the computation of the method's result and side effects. Those are coded by separate messages which are executed later on (see the description of *next* edges below). Only very simple methods consisting of one statement or expression only can be encoded into a single DPO rule.

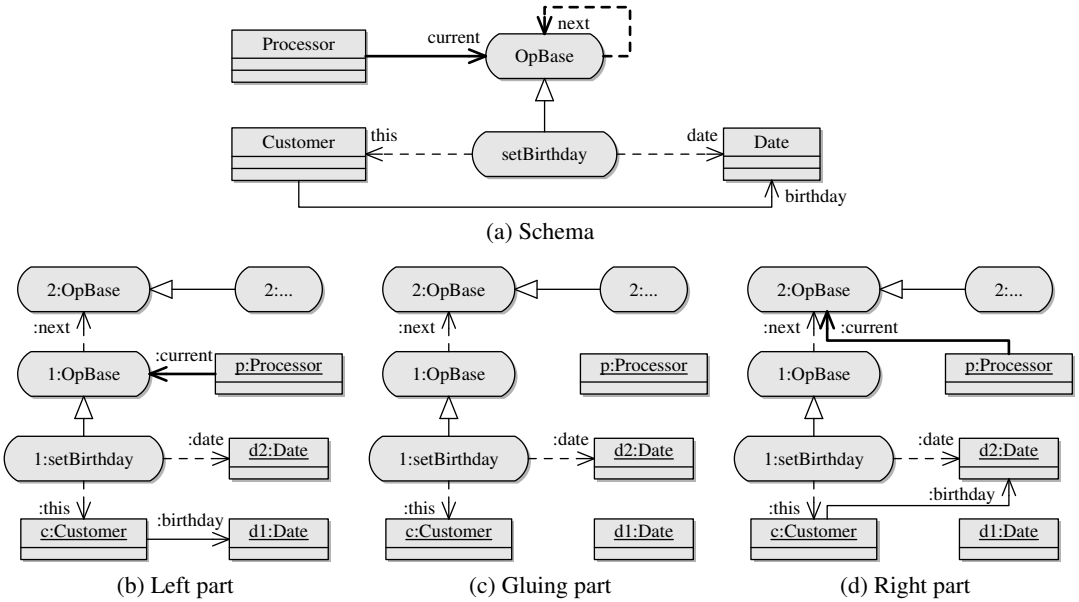
In order to be able to determine which message is ready to be processed, a special "marker" object called *processor* is used. A message referenced by a processor through a special *current* link is called *active*. Methods are formulated such that their left part requires an active message. Additionally, 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*.[20] Multiple processor objects can be used to model multi-threaded processing. Figure 6 displays the DPO rule for a method changing the target of a link. Note that analogously to classes, operations can also be specialised, and like objects, messages also possess a particle structure. This allows processor objects to refer to any message as well as to connect arbitrary messages via *next*.

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 [EEPT06]. 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 [EEPT06] 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 which fulfil all the axioms. The following figures demonstrate two such counter examples: adding a link violates axiom (M.19) (Fig. 7), and eliminating inheritance violates axiom (MP.3) (Fig. 8). 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.

In order to rule out situations as depicted in Fig. 7 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.

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

[20] 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 an *else* argument instead.



(a) Schema

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

Figure 6: Example method "Change target of *birthday* link"

Definition 3 (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

$$source^L(e_L) = l_N(k)$$
$$source^R(e_R) = r_N(k)$$
$$type_{L,E}(e_L) = type_{R,E}(e_R)$$

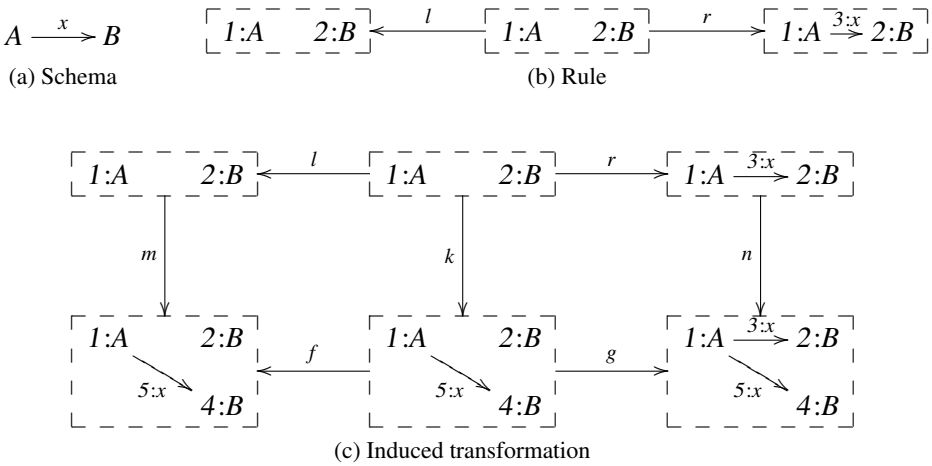
hold.

In order to rule out situations as depicted in Fig. 8, we restrict DPO rules to *completing* homomorphisms which "pull back" relations. Completing homomorphisms are an extension of strictly full homomorphisms:[21] 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. This is illustrated in Fig. 9 by referring to the predicate *under*. On the left side, the homomorphism f is not completing as the *under* predicate is not pulled back completely: There is no preimage for y' which is "under" x. On the right side, f has been made completing by adding the missing preimage y for y' such that y is "under" x.

This leads to the following definition:

Definition 4 (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

[21] 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 $p \in P_w$.



(a) Schema (b) Rule

(c) Induced transformation

Figure 7: Adding a link violates axiom (M.19) on page 8

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:

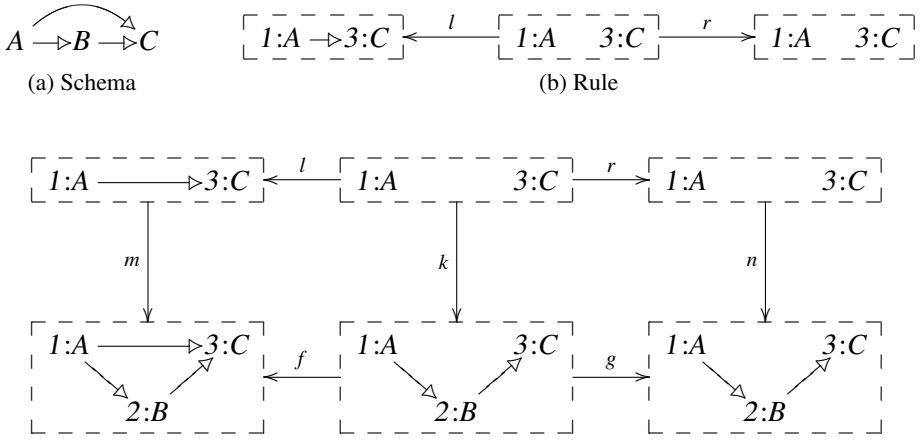
Definition 5 (Valid rule) A DPO rule $L \xleftarrow{l} K \xrightarrow{r} R$ is *valid* iff it pulls back edges and l and r are completing homomorphisms.

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 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.[22]

Now we can state the main theorem of this section:

Theorem 1 (Transformation preserves axioms [Sch09b, 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.*

[22] [Sch09b] shows in full detail how this can be done.

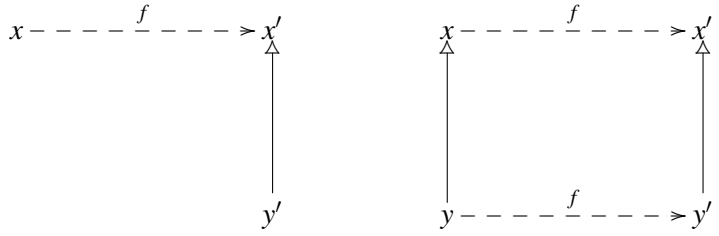


(a) Schema (b) Rule

(c) Induced transformation

Figure 8: Eliminating inheritance violates axiom (MP.3) on page 6

Proof sketch. The pushout complement D on the left side of a DPO transformation is a $\mathbf{Sys}(S)$-object because it is a subobject of G by construction, and implicational classes are closed under the formation of subobjects [AHS04, Cor. 16.19]. On the right side, we have to show for each M-axiom separately that its validity is retained by pushouts in $\mathbf{Alg}(MP^*)\downarrow S$. The validity of the homomorphism axioms and all schema-related axioms in H follows directly from the fact that H is an object in $\mathbf{Alg}(MP^*)\downarrow S$ and that S is an MP-system. Axiom (M.9) is valid in H because the pushout construction does not add any new elements to H as the underlying signature of MP^* consists of unary operation symbols only [EEPT06, Fact 8.12]. Axiom (M.17) is valid in H because for any pair of nodes $(x,y) \in \mathit{underI}^H$, there is a pair of preimages (x',y') related by underI in either R or G, due to the pushout morphisms being jointly surjective [AHS04, Prop. 11.29]. As both of these systems are MP-systems, they are related by rel due to axiom (M.17), and so are x and y, as all homomorphisms and especially the pushout morphisms preserve relations. A similar argumentation is used to prove that the axioms (M.15) and (M.18) are valid in H. The proof of the validity of the axioms (M.10), (M.11), and (M.16) needs more work. Here it is necessary to assume the second property of valid rules, namely that the rule's morphisms are completing. This ensures that in all cases, any node constellation in H that fulfils the premise of one of the three



(a) not completing (b) completing

Figure 9: Example of a non-completing and a completing homomorphism

axioms can be reflected to either R or G (or both), such that the same argumentation as above can be used to show the validity of the axioms in H. Last but not least, the proof of the validity of axiom (M.19) in H needs the first property of valid rules, namely that the span pulls back edges. This ensures that if one assumes two edges in H with the same source, both edges have preimages under the same pushout morphism. The rest of the proof follows similarly. □

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 processes (the migration of programs is handled in the next section).[23]

Definition 6 (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 (see Fig. 11 on page 16 for an example of folding and unfolding). Refactorings are special transformations which are constrained to surjective homomorphisms on the left side of the span. This constraint stems from the fact that refactorings are not allowed to delete schema objects because such a deletion almost always causes loss of information at the instance level, which contradicts the intuitive requirement that a refactoring should 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.[24]

First, we need a technical lemma which ensures that the epireflector $\mathscr{F}$ from Proposition 1 never changes the schema:

Lemma 3 (Epireflector $\mathscr{F}$ does not change schema) *Let $I \xrightarrow{type_I} S \in \mathrm{Ob}^{\mathbf{Alg}(MP)^2}$ be given, and let $\mathscr{F}_{\mathrm{Ob}}(I \xrightarrow{type_I} S)$ be the typed instance $I' \xrightarrow{type_{I'}} S'$. Then $S = S'$ holds.*

Proof. See [SLK10, Lemma 9]. □

Given a typed instance $I \xrightarrow{type_I} S$ and a schema transformation $t\colon S \overset{S^\#}{\rightsquigarrow} S'$, the migration is performed according to [SLK10] as follows (visualised in Fig. 10):

(1) $\mathscr{P}^{l^t}$, 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 for unfolding instance elements if l^t is not injective, and for deleting elements if l^t is not surjective.

[23] See [LKSP06a, LKSP06b, KLS07] for precursor material on data migration induced by schema transformations.

[24] Schema transformations such as deletion of classes without instances are not considered to be refactorings since our classification is purely schema-based. We never look at the instance level in order to classify a schema transformation as information-preserving or (possibly) information-destroying.

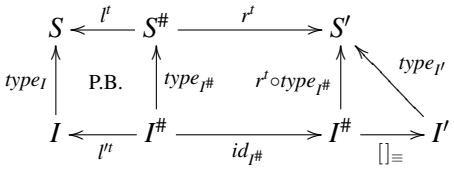


Figure 10: Schema transformation and instance migration

(2) $\mathscr{F}^{r^t}$, 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. In order to fix this, we apply the epireflector $\mathscr{F}\colon \mathbf{Alg}(MP)^2 \to \mathbf{Sys}$ to it, obtaining the typed instance $I' \xrightarrow{type_{I'}} S'$ in the subcategory $\mathbf{Sys}(S')$. This part of the transformation is responsible for identifying instance elements due to retyping.

Note that due to Lemma 3, the schema is left unchanged, so the epireflector can be restricted to the slice category for a given schema S, yielding the functor $\mathscr{F}^S\colon \mathbf{Alg}(MP)\!\downarrow\! S \to \mathbf{Sys}(S)$. The composition of the three functors above results in the migration functor defined below:

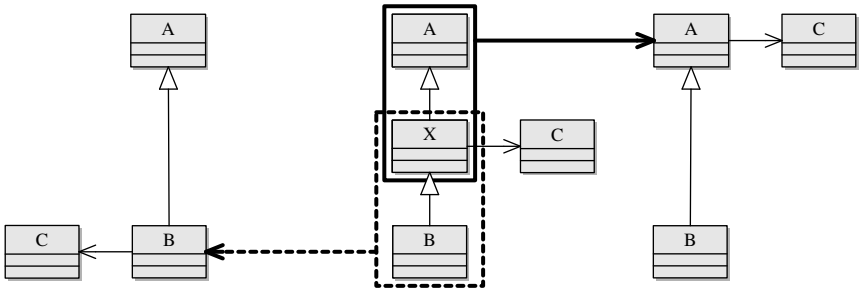
Definition 7 (Migration functor) Let $t\colon S \overset{S^\#}{\rightsquigarrow} S'$ be a transformation. The *migration functor* $\mathscr{M}^t\colon \mathbf{Sys}(S) \to \mathbf{Sys}(S')$ is defined as:

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

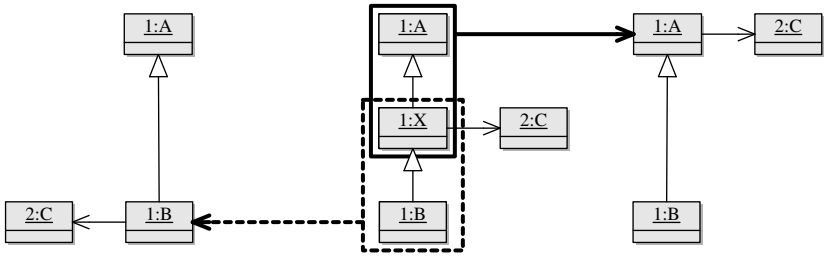
where the functor $\mathscr{P}^{l^t}\colon \mathbf{Sys}(S) \to \mathbf{Alg}(MP)\!\downarrow\! S^\#$ is the pullback functor along l^t, the functor $\mathscr{F}^{r^t}\colon \mathbf{Alg}(MP)\!\downarrow\! S^\# \to \mathbf{Alg}(MP)\!\downarrow\! S'$ is the composition functor along r^t, and $\mathscr{F}^{S'}\colon \mathbf{Alg}(MP)\!\downarrow\! S' \to \mathbf{Sys}(S')$ is the epireflector $\mathscr{F}\colon \mathbf{Alg}(MP)^2 \to \mathbf{Sys}$ restricted to the slice category $\mathbf{Alg}(MP)\!\downarrow\! S'$.

The example in Fig. 11 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 (M.18) is satisfied.[25] Similar transformations which manipulate the schema graph by moving the source or the target of an association, an attribute, or an inheritance edge can be formulated and induce corresponding migrations on the instance level.

[25] Note that the effect shown cannot be simulated by deleting the association on the left and re-adding it on the right because the induced migration leads to data loss: the middle instance graph will have all links to the deleted association removed, and the right instance graph will be identical to the middle one as the schema addition will have no effects on the instance level.



(a) Schema transformation



(b) Induced migration

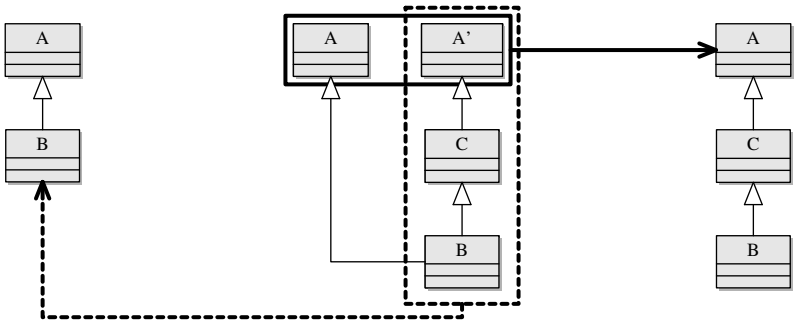
Figure 11: Moving the origin of an association upwards the inheritance hierachy

Another type of transformations based on unfolding and folding is introducing a class. This is illustrated in Fig. 12. Note that this transformation can easily be adapted to the situation where *B* has no existing superclasses (thereby introducing a new root class *C*) by removing the classes *A* and *A'*, making the folding on the right side the identity operation.

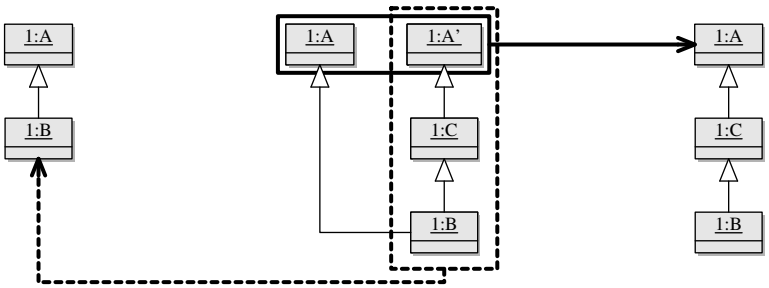
6 Method Migration

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. 5, 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. 13, 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 elementary properties of pushouts (see e. g. [EEPT06]).

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 transformation



(b) Induced migration

Figure 12: Introducing a class

Definition 8 (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 2 (Migration preserves valid DPO rules [Sch09b, Proposition 15.23]) *Given a proper transformation* $t\colon S \overset{S^*}{\rightsquigarrow} S'$, *let*

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

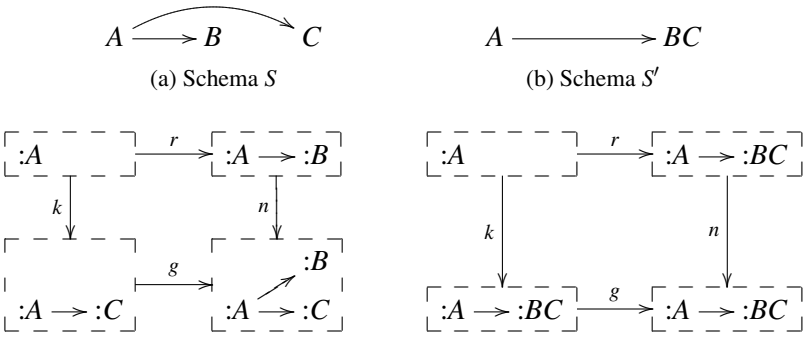
be a valid DPO rule. Then

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

is a valid DPO rule as well.

Proof sketch. It has to be shown that the pullback functor and the epireflector both preserve valid DPO rules, i. e. completing homomorphisms and spans pulling back edges. For the pullback functor, the preservation of both properties is proven easily by using the pullback properties that pullback morphisms are jointly monic and that certain unique preimages exist in the pullback object.[26] To show that the epireflector preserves spans pulling back edges is done by a sort

[26] If $(f'\colon P \to B, g'\colon P \to A)$ is pullback of $(f\colon A \to C, g\colon B \to C)$, and if $f(x) = g(y)$ for some x and y, then there



(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

Figure 13: Pushout in $\mathbf{Alg}(MP^*)\downarrow S$ is not preserved by a migration

of "triple pull-back": first, the edge under consideration is pulled back along the surjective reflection arrow, then it is pulled back along the valid DPO rule, and finally it is transferred to the epireflective subcategory again by applying the epireflector on it. The remaining case that the epireflector preserves completing homomorphisms is proven similarly, but has to be extended as simply pulling back along the surjective reflection arrow may not suffice due to predicates being made true by the epireflector.[27] So for each predicate and each axiom that may render this predicate true, the triple pull-back approach described above has to be proven separately, taking the axiom's conclusion into consideration. □

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

Proposition 3 (Migration preserves pushouts [Sch09b, Proposition 15.35]) *Let* $t\colon S \overset{S^*}{\rightsquigarrow} S' \mathrel{\widehat{=}} S \xleftarrow{l'} S^* \xrightarrow{r'} S'$ *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*

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

is a pushout of

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

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

exists a unique z in the pullback object P with $g'(z) = x$ and $f'(z) = y$. This follows directly from the limit property of pullbacks by comparing the pullback square with the terminal object.

[27] This includes the identification of elements, as for each sort, there is a corresponding equality predicate.

Proof sketch. First it is proven that the pullback functor preserves DPO pushouts. First note that DPO pushouts are pullbacks in our categories as the DPO rule morphisms are completing and, as such, injective [EEPT06, Remark 2.25]. So applying the pullback functor to a DPO pushout yields a pullback in the target category, as the pullback functor preserves limits due to being right-adjoint to the composition functor along the same morphism [Gol84, section 15.3]. It remains to show that the resulting pullback diagram is also a pushout. This is done by proving that the morphisms in the target diagram meet certain conditions; in particular, if $(f'\colon P \to B, g'\colon P \to A)$ is pullback of $(f\colon A \to C, g\colon B \to C)$, then this square is a pushout if f and f' are injective, if f and g are jointly surjective, and if g is injective up to f'[28] (compare again [EEPT06, Remark 2.25]).

For the second part of the proof, note that applying the epireflector to a pushout diagram necessarily yields a pushout diagram in the target category because the epireflector is left-adjoint to the inclusion functor. However, we have to show that this pushout diagram in the subcategory $\mathbf{Sys}(S')$ is also a pushout diagram in the category $\mathbf{Alg}(MP_*)\downarrow S'$, as this is the category where DPO transformations take place. To show this, we first prove that this is true if the following compatibility condition is met. Let $(f'\colon B \to P, g'\colon A \to P)$ be pushout of $(f\colon C \to A, g\colon C \to B)$ in $\mathbf{Alg}(MP_*)\downarrow S'$. Then applying the epireflector to this diagram yields again a pushout in the same category if for each predicate $p \in P_w$ and each tuple $x \in ker_p^{u^P}$ we have some $y \in ker_p^{u^A}$ with $g'_w(y) = x$ or some $z \in ker_p^{u^B}$ with $f'_w(z) = x$, i. e. if the kernel of the reflection arrow u^P can be separately described by the kernels of the reflection arrows u^A and u^B (without any interactions between these kernels). Having proven this, we show for each predicate that this property is fulfilled by the epireflector into $\mathbf{Sys}(S')$. □

Both propositions can be combined, yielding the following theorem:

Theorem 2 (Correctness of the migration of programs [Sch09b, Theorem 15.36]) *Let $t\colon S \overset{S^*}{\rightsquigarrow} S'$ be a proper transformation. Then the migration functor $\mathscr{M}^t$ 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 2 and Proposition 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 $\mathscr{P}^{l^t}$ is right-adjoint where the second factor—the composition functor $\mathscr{F}^{r^t}$—is its left-adjoint. But the third factor—the construction $\mathscr{F}^S$ 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.

[28] This means that $g(x) = g(y)$ with $x \neq y$ implies $x, y \in \operatorname{Im} f'$.

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* 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 2 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 [Gol84, EEPT06]. An approach can be found in [KLS07] which covers data migration only. Hence, an extension to method migration is desirable.

References

[AHS04] J. Adámek, H. Herrlich, G. E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. Free Software Foundation, 2004. http://katmat.math.uni-bremen.de/acc/acc.pdf

[BEL+03] R. Bardohl, H. Ehrig, J. de Lara, O. Runge, G. Taentzer, I. Weinhold. Node Type Inheritance Concept for Typed Graph Transformation. Technical report 2003-19, Technical University, Berlin, 2003. http://user.cs.tu-berlin.de/~rosi/publications/BELRTW03_TR03-19.ps.gz

[CDFR04] A. Corradini, F. L. Dotti, L. Foss, L. Ribeiro. Translating Java Code to Graph Transformation Systems. In *Proceedings of the 2nd International Conference on Graph Transformation (ICGT 2004)*. Pp. 383–398. 2004.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[FS03] M. Fowler, K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[Gol84] R. Goldblatt. *Topoi: The Categorical Analysis of Logic*. Dover Publications, 1984.

[KKR06a] H. Kastenberg, A. G. Kleppe, A. Rensink. Defining Object-Oriented Execution Semantics Using Graph Transformations. In Gorrieri and Wehrheim (eds.), *Proceedings of the 8th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems (FMOODS 2006)*. Lecture Notes in Computer Science 4037, pp. 186–201. Springer-Verlag, June 2006.

[KKR06b] H. Kastenberg, A. G. Kleppe, A. Rensink. Engineering object-oriented semantics using graph transformations. Technical report TR-CTIT-06-12, University of Twente, Department of Computer Science, 2006.
http://www.cs.utwente.nl/~kastenbe/papers/taal.pdf

[KLS07] H. König, M. Löwe, C. Schulz. Functor Semantics for Refactoring-Induced Data Migration. Technical report 02007/01, Fachhochschule für die Wirtschaft Hannover, 2007.
http://fhdwdev.ha.bib.de/docmgr/index.php?module=fileview&objectId=95

[LBE+07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Attributed Graph Transformation with Node Type Inheritance. *Theoretical Computer Science* 376(3):139–163, 2007.

[LKSP06a] M. Löwe, H. König, C. Schulz, M. Peters. Refactoring Information Systems – A Formal Framework. In *Proceedings of the 10th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2006)*. Volume 1, pp. 75–80. 2006. Also appeared in: Journal on Systemics, Cybernetics and Informatics, 5(2):66–71, 2007.

[LKSP06b] M. Löwe, H. König, C. Schulz, M. Peters. Refactoring Information Systems – Handling partial composition. *Electronic Communications of the EASST* 3, 2006.

[Mal73] A. I. Mal'cev. *Algebraic systems*. Springer-Verlag, 1973.

[MT04] T. Mens, T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions On Software Engineering* 30(2):126–139, 2004.

[Sch09a] C. Schulz. Mehrsortige algebraische Systeme. Technical report 02009/06, Fachhochschule für die Wirtschaft Hannover, 2009.
http://fhdwdev.ha.bib.de/docmgr/index.php?module=fileview&objectId=429

[Sch09b] C. Schulz. Refactoring objektorientierter Systeme. Technical report 02009/02, Fachhochschule für die Wirtschaft Hannover, 2009.
http://fhdwdev.ha.bib.de/docmgr/index.php?module=fileview&objectId=282

[SLK10] C. Schulz, M. Löwe, H. König. Categorical Framework for the Transformation of Object-Oriented Systems: Models and Data. Technical report 02010/01, Fachhochschule für die Wirtschaft Hannover, 2010. Submitted for publication in: Journal of Symbolic Computation.
http://fhdwdev.ha.bib.de/docmgr/index.php?module=fileview&objectId=525

[Wec92] W. Wechler. *Universal Algebra for Computer Scientists*. Springer-Verlag, 1992.