Formal and tool-supported operator for multi-formalism modelling

Jonathan Marchand

Formal and tool-supported operator for multi-formalism modelling

Internship report

Jonathan Marchand
under the direction of Benoît Baudry and Benoît Combemale
Triskell team
IRISA

Master Recherche en Informatique, Université de Rennes 1
ENS Cachan, Antenne de Bretagne

Abstract

We propose in this master internship report a unified formal framework for model composition. Indeed, there are various operators of model composition defined each in its own framework. Defining a unified framework for model composition allows to compare them, to reuse them and to capitalize knowledge. Defining a formal framework allows to consider the system that these compositions represent in its entirety. This framework is built through category theory and the notion of pushout. More precisely, a category of models is defined, and the pushout construction in this category described. Then this pushout is used to formalize several model composition operators, including model merge, model transformation and aspect weaving.

Contents

1 Introduction

In order to tackle the growing complexity of nowadays software projects, model driven engineering (MDE) defines a framework and a set of good practises. The basis of MDE is the use of a high level of abstraction, the models, to reason and solve design problems. This notion of models lead to the idea of separation of concerns: as a model represents only a view of the system, each concern is modeled separately in a different model. MDE advocate the use of domain specific modeling languages (DSML) to write these models. A DSML is a combination of a syntax and a semantic adapted to the expression of the notions attached to a concern. Thus, the work can then be easily distributed over several teams, each team working on a specific concern with adapted tools.

The notion of model composition arises naturally in this context: we have several models, but only one system to model [3]. Hence we need ways to compose all these models for different purposes, e.g. create a model of the system to feed a code generator or check the consistency of the models. In fact, the notion of model composition includes many operations on models, e.g. model merge, model transformation or aspect weaving. However, the notion of model composition is not itself clearly defined, preventing the possibility of capitalizing knowledge from different model compositions.

To have a better understanding of the notion of model composition, we need to define a unified, formal framework for model composition. Its unified nature will allow us to compare compositions operators, to capitalize knowledge and to reuse it to define new composition operators. The framework's formal nature is essential to extract properties on all the system that is modeled. Indeed, the compositions give properties on the models they compose, so on the system. Furthermore, the formalism is essential to make the most of multi-formalism modeling, allowing to work with several different DSML.

Category theory [13] appears as a good candidate for such a framework. Indeed, its high level of abstraction allows the identification of different notions in different fields under a common vocabulary. More precisely, the categorical notion of pushout, which behaves like a merge when applied to graphs [19], seems to be a good candidate to formalize several operators of composition. Moreover, this pushout has a constructive definition, which makes it usable in practice. We have therefore defined a category of models and used the pushout to define several model composition operators.

We begin this report by a state of the art, in order to set this work in the context of model driven engineering and to set what we mean by "model composition". Then we define some category theory notions, which are used to define a category of models. Finally, we use this category to formalize several operators of model composition.

2 State of the art

2.1 Model driven engineering

Model driven engineering (MDE) is a paradigm which aims at tackling the growing complexity of software projects by describing a system at a high-level of abstraction. This description, the model, enables the designers to ignore the complexity of the reality and to concentrate on some of its aspects only. There are many definitions of a model. We will use the following one: "*To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A.*"([14]).

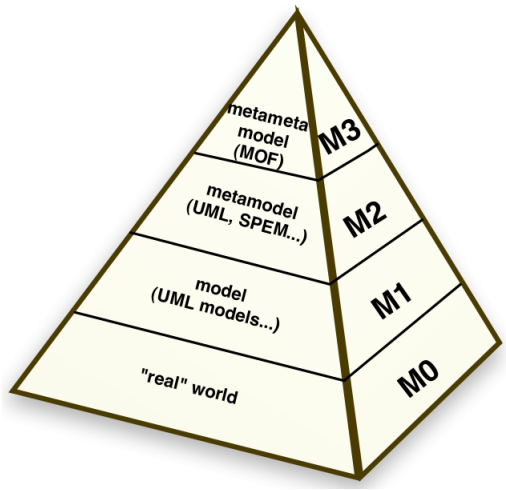


Figure 1: OMG modeling pyramid

From this definition, we can raise two characteristics of a model. First, a model *represents* a system. This representation notion is quite informal, as it is very difficult to determine if a model represents well a system. A lead is in the second characteristic of a model: it must be able to answer several questions about the system just in the same way the system would have, no more, no less. This is the substitutability principle ([14]).

As the modeled system may be very complex, a complete model, which would answer all the questions, would be very complex too. This must be avoided, as the simpler a model is, the easier it is to think about it. So, the system is described through several, not very complex models. Each model addresses a particular problem, enabling to separate the concerns. This is particularly useful when the work is split over several teams: each team models only its concerns, and does not get superfluous information. As each model addresses a different concern, each model is described in its own Domain Specific Modeling Language (DSML). This allows to use the most suited notions to describe a concern. This idea of using as much DSML as needed is a main idea in MDE, and what makes it different from the popular UML approach, which is quite monolithic.

As in the MDE view, everything is a model, those DSML are logically defined through models which are called *metamodels*. A metamodel is described using a meta-metamodel, the Meta-Object Facility (MOF). The MOF is *meta-circular*, meaning that it can model itself, and thus do not need the introduction of any additional level of abstraction. Moreover, the MOF gives a general framework for the description of the metamodels. The MOF has been defined by the Object Management Group (OMG) in [15]. This yields the OMG modeling organisation, represented as a pyramid on figure 1. This organisation is in fact quite natural, as underlined e.g. by a comparison with the grammar world: a program execution (M0) conforms to a program code (M1) which conforms to a language grammar (M2) which conforms to Extended Backus-Naur Form (EBNF) (M3) which describes itself.

2.2 Model composition

As we have seen in the previous section, in the MDE process, several models are designed to describe a single system. However, they describe one system, and model composition may be needed to check for consistency between those models or simply feed a code generator. Thus, model composition is an essential concern in MDE.

Though, several problems are inherent to model composition. Indeed, the models to be composed are often expressed in several different DSML, expressing different properties, and the resulting model will itself be described in a DSML, which may be unable to express some of these properties. Moreover, a same concept (e.g. Person) may have been named differently among the different models (e.g. Client, Buyer and Purchaser), making difficult the use of automatic tools, not talking of the properties attached to these concepts. Another problem is that, in the general case, the composition is not commutative, or even associative. Thus, given a set of models representing a system, we cannot compose them in any order, which would have been convenient.

In the following of this subsection, we introduce several model composition operators.

2.2.1 Model merge

Model merge is an operation which takes as inputs two or more models, a mapping between them, and gives as output a merged model [3]. The merged models must include every element of the input models [17]. The mapping indicates which elements of the input models represent the same concepts, and thus should be represented only once in the merged model. The mapping may be implicit, as in UML merge [9, 10], where elements with the same name are considered as mapped together.

An example of merge is pictured on figure 2. Two models, containing respectively the classes **Person** and **Customer**, are linked by a mapping, represented with dashed edges. The mapping indicates that **Person** and **Customer** represent the same notion, and thus should be unified in the merge, as well as the fields "name".

In fact, the mapping may be much more complex, including notions such as equality and similarity [1]. If two elements are related by a similarity link, they are supposed to be linked someway in the output model, whereas an equality link simply implies a merge. We do not treat such notions in the following.

The question of creating the mapping is also quite complex. It can either be provided by a domain expert, generated automatically based on similarities, like the names of the elements, or, most likely, be the result of semi-automatic process. For the purpose of our study, we suppose that the mapping is provided.

2.2.2 Model transformation

A model transformation is defined by a set of transformation rules, which are applied to the model to transform. More specifically, we may distinguish endogenous and exogenous transformations, the former modifying the model in place, while the latter translate the model to another DSML. A rule specifies the pattern to match in the model, and how it is transformed. In the case of endogenous model transformation, it says if some elements are added or deleted, and which elements of the pattern are not modified. The rules may be applied randomly until none of them matches the model, or applied in a specific order, following an algorithm.

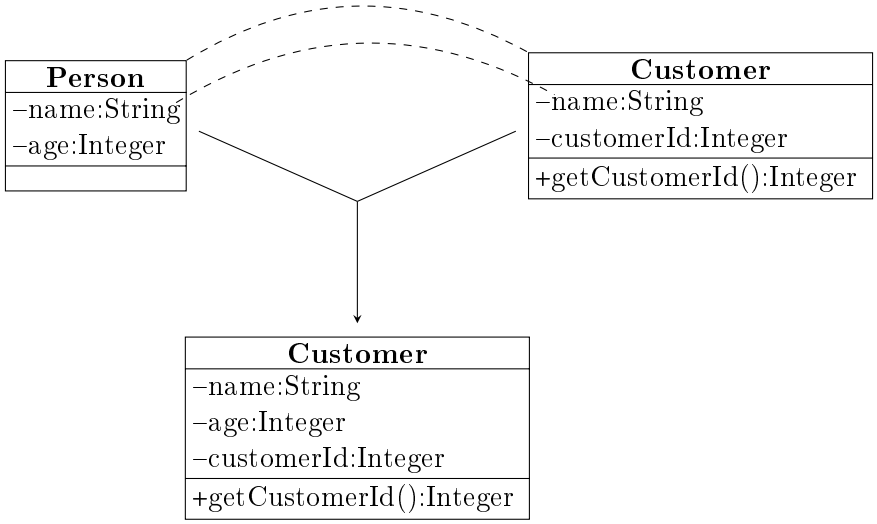


Figure 2: Two models linked by a mapping and their merge.

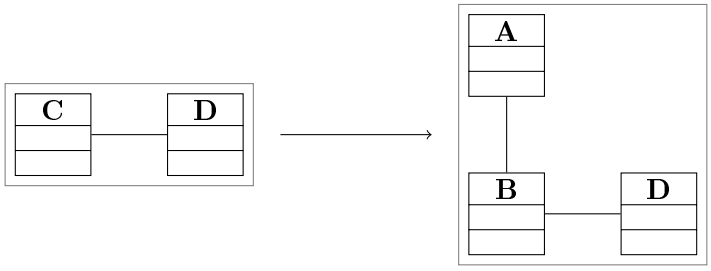


Figure 3: An example of endogenous transformation rule.

An example of endogenous transformation rule is pictured on figure 3. The left part pictures the pattern to match, while the right part pictures how the pattern should be transformed. The arrow between the two is in fact a relation linking the two D, indicating that this element is preserved during the transformation. The class C is deleted, while A and B are created. A link between B and D is also created.

2.2.3 Aspect weaving

Aspect-oriented programming [12] (AOP) is a paradigm which aims at separating concerns. It introduces the notion of aspect, which is the modularization of a cross-cutting concern. An aspect encapsulate all the code related to a concern, but which is scattered in the program code, and how it is supposed to be integrated within the code. In practice, there is a base program, and additional, cross-cutting functions encapsulated into aspects.

There are two types of aspect weaving: structural and dynamic. Dynamic aspect weaving detects, at run-time, patterns of execution in the traces and start new behaviours. Structural aspect weaving "re-open" the base program code and add everything the aspects specifies. In the following, we only address structural aspect weaving.

The notion of aspect can be applied to modeling [11]. An aspect is composed of a set of pairs pointcut/advice: the pointcut gives the pattern to match in the base model, the advice gives the elements to add.

An example of aspect weaving is pictured on figure 4. On the top, the couple pointcut-advice. On the bottom, the base model and the result of the weaving on the base model. Here the goal of the aspect is to add behaviour to a finite state machine (FSM). The pointcut just matches the FSM structure, without taking care of details such as whether or not the states have names, and the advice add everything needed for adding behaviour: an initial and a current state, and several methods.

2.2.4 Synchronization

As said before, a system is modeled through many different models to ensure separation of concerns. However, the different models evolve during the system lifetime, and we want to keep them synchronized (see figure 5). Synchronization consists in propagating the updates made to a model to the other models.

2.3 Category theory

Category theory has been introduced in the 40s by Eilenberg and Mac Lane to solve algebraic topology problems [6]. However, it quickly appeared that category theory allowed to reason on similar notions in various fields of mathematics. This is due to its high level of abstraction: as category theory only deals with objects and arrows between this objects, it can describe a great variety of situations under a common language, and allow to capitalise theorems.

Due to this extreme versatility, category theory has also been used in computer science, notably in type theory, but also in software engineering. For example, it has been used for a long time in the field of graph transformation [18] through the double pushout[1] and the single pushout approaches. It has been transposed to model transformation in the EMFTiger tool [2]. These transformations use the properties of the pushout, a categorical construction (see 3.5). The framework was later extended to deal with information preservation [4], and synchronization [7]. Though, this framework has a serious drawback: as it was first designed for graph transformations, it does not allow us to take totally into account the specificity of models (e.g. references, which are more than just edges), though some efforts have been done in this direction through the introduction of typed attributed graph transformation [5].

Another approach was to use pushouts as a way to merge models. This approach has been encouraged since the 90s by Goguen [8] and applied to merge models and check their consistency [21]. Indeed, the intuition behind pushouts is that they put structures together without adding nor deleting anything [20]. It is this idea that seems us interesting to formalize model composition and lead us to examine pushout, though in the above examples of model composition, we also delete elements. We overcome this difficulty through the use of partial functions, as explained in section 4.4.

3 Mathematical preliminaries: Category theory

As seen in section 2.3, the pushout has interesting properties for what we want to do. In order to use it formally, we define what is a category, and then what are the coproduct and the

[1] which leads to the so-called triple graph grammars (TGG).

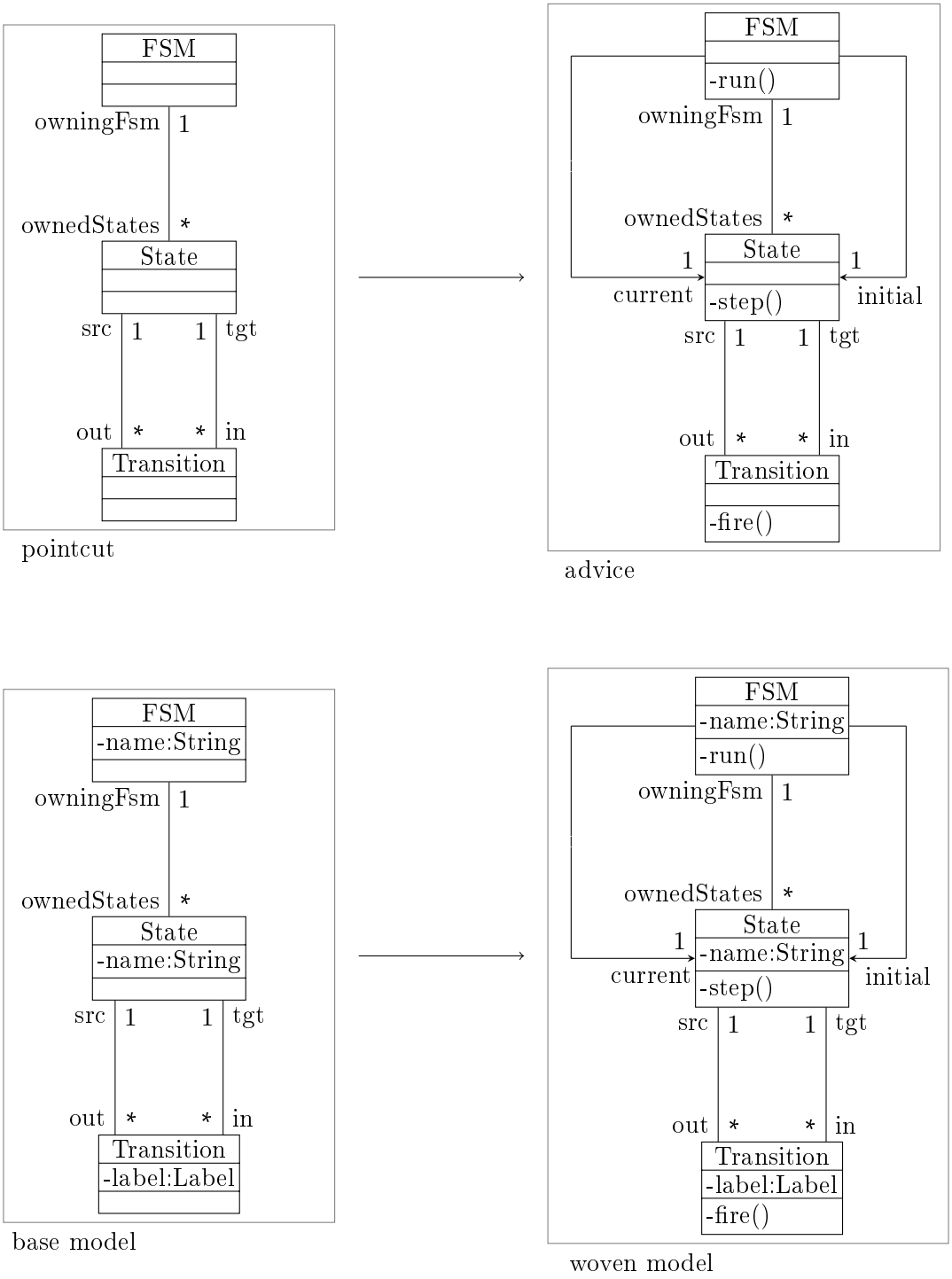


Figure 4: An example of aspect weaving. On the top, the couple pointcut-advice. On the bottom, the base model and the model with the aspect woven. Here the goal of the aspect is to add behaviour to a Finite State Machine.

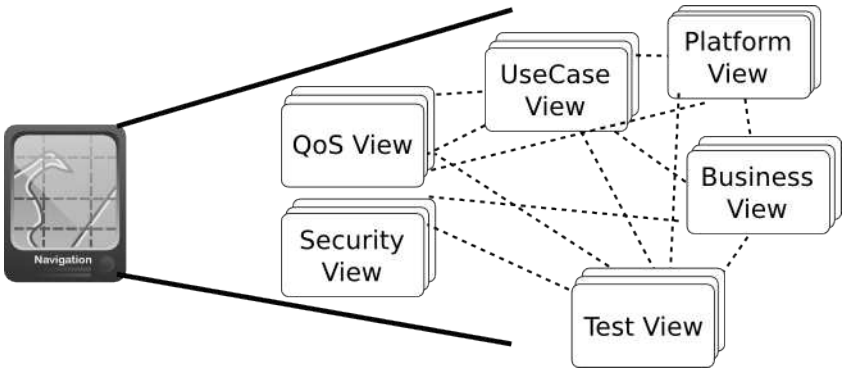


Figure 5: A GPS system modeled through many views for the purpose of separation of concerns. Dashed lines represent model synchronization.

coequalizer, which are used in a constructive definition of the pushout.

3.1 Definition

Definition 1 (Category)**.** A category consists of:

- a collection of objects
- a collection of arrows

with the following properties:

- Each arrow has a source and a target, called respectively domain and codomain. We note $f : A \to B$ to show that the arrow f has for domain A and for codomain B.
- There is a composition operator for arrows, such that for every $f : A \to B$ and $g : B \to C$, there exists $g \circ f : A \to C$ in the category.
- This composition is associative: $h \circ (g \circ f) = (h \circ g) \circ f$, as shown in figure 6.
- To each object A is associated an identity arrow $id_A : A \to A$, which is neutral for the arrow composition, i.e. $\forall f : A \to B, id_B \circ f = f = f \circ id_A$

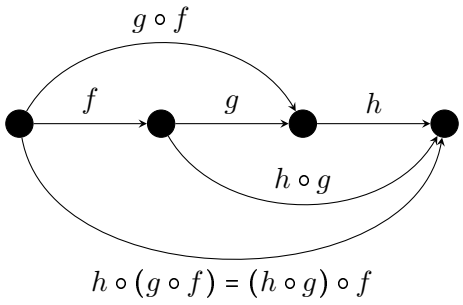


Figure 6: Associativity of the arrow composition

Remark. Computer scientists sometimes write $f;g$ instead of $g \circ f$. We use the mathematical notation (with $\circ$) as it is the most frequent, even in computer science.

In the rest of this section, we consider two examples of categories: the category **Set** of sets and the category **Graph** of graphs. The goal is not to prove formally all the statement we do about this category, but rather to give an intuition of the different notions we introduce. All the results given about the category **Set** can be found in [16], and those of the category **Graph** can be derived from the notion of comma category [19].

Example (The category **Set**)**.** The category **Set** has the sets as objects and the total functions[2] as arrows. The composition of arrows is the usual function composition. The identity arrows are the identity functions.

We check that **Set** is a category:

- Every arrow has a domain and a codomain, which are the usual domain and codomain of functions.
- The composition of two total functions $f : A \to B$ and $g : B \to C$ is the total function $g \circ f : A \to C$ mapping every element $a \in A$ on $g(f(a))$.
- The composition of functions is associative.
- The identity functions are neutral for function composition.

Example (The category **Graph**)**.** Formally, a graph is a pair $G = (V, E)$ with V a set of vertex and $E \subseteq V \times V$ a set of edges. For $e = (v_1, v_2) \in E$, we note $fst(e) = v_1$ and $snd(e) = v_2$. A graph morphism $h : A \to B$ is a couple of functions $h = (h_V, h_E)$, with $h_V : V_A \to V_B$ and $h_E : E_A \to E_B$, which preserves the structure of the graphs, i.e. $\forall e \in E_A, fst(h_E(e)) = h_V(fst(e))$ and $snd(h_E(e)) = h_V(snd(e))$.

The category **Graph** has graphs as objects and graph morphisms as arrows. Composition of arrows is the morphism composition, identity arrows are identity morphisms.

Graph is a category, as:

- Every morphism has obviously a domain and a codomain
- The morphism composition is defined component-wise: the composition of $f : A \to B$ with $g : B \to C$ is the graph morphism $g \circ f = (g_V \circ f_V, g_E \circ f_E)$ from A to C mapping each element $a \in V_A$ to $g_V(f_V(a))$ and each element $e \in E_A$ to $g_E(f_E(e))$.

  We check that $g \circ f$ is well-defined: as g and f are morphisms, for every $e \in E_A$, $fst(g_E(f_E(e))) = g_V(fst(f_E(e))) = g_V(f_V(fst(e)))$, the same for snd, so $g \circ f$ is a morphism.
- The composition is associative, as it is done component-wise and the composition of functions is associative:

$$\begin{aligned} h \circ (g \circ f) &= (h_V \circ (g_V \circ f_V), h_E \circ (g_E \circ f_E)) \\ &= ((h_V \circ g_V) \circ f_V, (h_E \circ g_E) \circ f_E) \\ &= (h \circ g) \circ f \end{aligned}$$

- The identity morphism $id_G = (id_V, id_E)$ is, for the same reason, neutral for composition.

[2] A *total* function corresponds to the usual notion of function. We nevertheless emphasize the total nature of these functions, in opposition with the partial functions (functions which are not defined on all their domain) that we use in section 4.

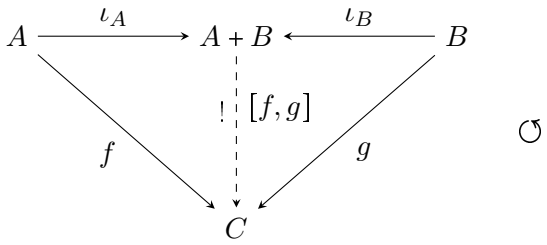


Figure 7: $(A + B, \iota_A, \iota_B)$ is the coproduct of A and B.

3.2 Coproduct

The coproduct is a generalization of the set-theoretic disjoint union. In set theory, the disjoint union is like the classic union, except that if the two sets have a non-empty intersection, the elements of this intersection are added twice in the disjoint union. In other words, we consider the two sets as disjoint. More formally, $A \uplus B \cong (A \times \{1\}) \cup (B \times \{2\})$.

Definition 2 (Coproduct)**.** A coproduct of two object A and B is an object $A + B$ together with two injections arrows ι_A, ι_B such that for any object C with $f : A \to C$ and $g : B \to C$, there exists a unique arrow $[f, g] : A + B \to C$ such that $[f, g] \circ \iota_A = f$ and $[f, g] \circ \iota_B = g$.

This definition is pictured on figure 7. The symbol ↺ on the right means that the diagram commutes, i.e. that the two equalities of the definition are satisfied: $[f, g] \circ \iota_A = f$ and $[f, g] \circ \iota_B = g$. The fact that the arrow $[f, g]$ is dashed emphasizes the fact that its existence is a consequence of the rest of the diagram. The exclamation mark "!" emphasizes the uniqueness of the arrow $[f, g]$, in the sense that it is the only arrow which makes the diagram commute.

Remark. The coproduct of two objects A and B is unique up to isomorphism. Loosely speaking, the uniqueness up to isomorphism is similar to the uniqueness up to renaming: all the coproducts of A and B have basically the same structure. Thanks to this uniqueness, we may talk of *the* coproduct of two objects, instead of *a* coproduct.

Example (In the category **Set**)**.** In **Set**, the coproduct of two sets A and B is $A \uplus B$, the disjoint union of A and B. The injection arrows are the canonical injection functions: $\iota_A : \begin{array}{ccc} A & \to & A \uplus B \\ a & \to & a \end{array}$.

Example (In the category **Graph**)**.** A graph is composed of two sets, a set of nodes and a set of edges, and the coproduct in the category **Set** of sets is the set-theoretic disjoint union, therefore we say that the coproduct in **Graph** is the component-wise disjoint union and we use the usual notation for disjoint union: $A \uplus B = (V_A \uplus V_B, E_A \uplus E_B)$, $\iota_A = (\iota_{V_A}, \iota_{E_A})$ and $\iota_B = (\iota_{V_B}, \iota_{E_B})$. The proof that it is indeed a coproduct is tedious but straightforward, using the facts mentioned above and the fact that the morphism composition is done component-wise.

The figure 8 pictures a coproduct in **Graph**. The intuition behind the coproduct is a generalization of the set-theoretic disjoint union: we put the two graphs together and use the injection functions as coloration functions, enabling us to distinguish from which graph comes each element in the coproduct. Thus, if we have two graph morphisms $f : A \to C$ and $g : B \to C$, the arrow $[f, g]$ induced by the coproduct properties is a pattern-matching on

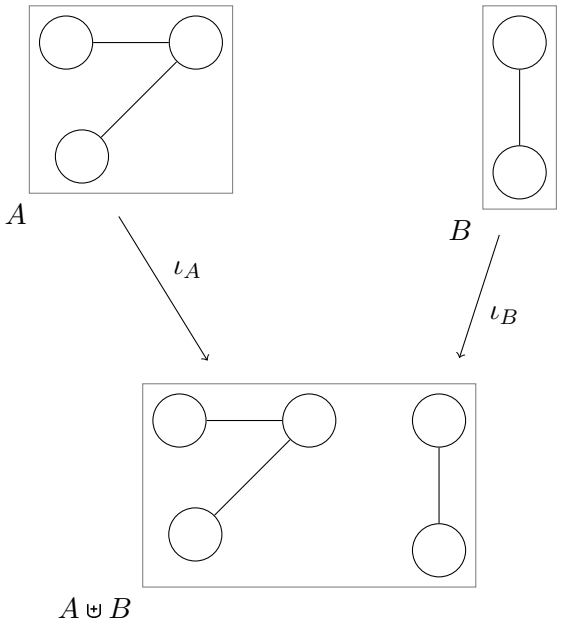


Figure 8: Coproduct in the category **Graph**.

$A \uplus B$: if an element comes from A (i.e. he is in the image of ι_A), we apply f, otherwise we apply g.

3.3 Coequalizer

The coequalizer is a generalization of the set-theoretic notion of quotient by an equivalence relation, or, in other words, of the notion of set of equivalence classes.

Definition 3 (Coequalizer)**.** A coequalizer of two arrows $f : A \to B$ and $g : A \to B$ is an arrow $h : B \to C$ such that $h \circ f = h \circ g$ and such that for every $h' : B \to C'$ with $h' \circ f = h' \circ g$, there exists a unique arrow $k : C \to C'$ such that $k \circ h = h'$.

This definition is pictured on figure 9. The graphical notations have the same meaning as for the coproduct.

Remark. The coequalizer of two objects is unique up to isomorphism, so, as for the coproduct, we may say *the* coequalizer of A and B, instead of a coequalizer of A and B.

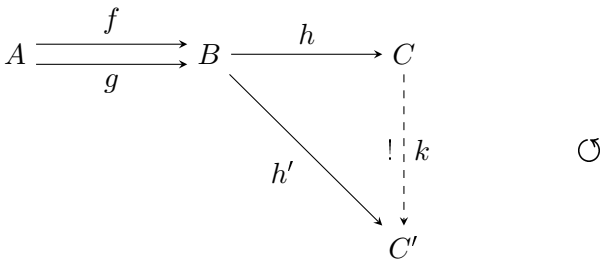


Figure 9: C is the coequalizer of f and g.

Example (In the category **Set**)**.** In the category **Set** of sets, the coequalizer of two functions $f : A \to B$ and $g : A \to B$ is given by the following construction. Let $\sim$ be the smallest equivalence relation such that $\forall a \in A, f(a) \sim g(a)$. Let $C = B/\sim$ be the quotient set of B by $\sim$, or, in other words, the set of equivalence classes of B under $\sim$. The canonical projection π from B to C, which maps every element b of B on its equivalence class $[b]$ in C, is a coequalizer of f and g.

Example (In the category **Graph**)**.** As for the coproduct, we construct the coequalizer of two morphisms $f : A \to B$ and $g : A \to B$ component-wise. More precisely, let $\sim_V$ be the smallest equivalence relation such that $\forall v \in V_A, f_V(v) \sim_V g_V(v)$, let $V_C = V_B/\sim_V$ be the quotient of V_B by $\sim_V$, and let $\pi_V : V_B \to V_C$ be the canonical projection function. $\sim_E$, $E_C = E_B/\sim_E$ and π_E are defined similarly. Let C be the graph (V_C, E_C), the morphism $\pi = (\pi_V, \pi_E)$ is a coequalizer of f and g.

This definition of the coproduct of two morphisms needs some verifications. Indeed, it is not obvious that C is a graph, neither that π is a morphism, nevertheless, except for these details, the proof that it is a coequalizer is tedious but straightforward, as for the coproduct.

The fact that C is a graph comes from the fact that graph morphisms preserve the structure of the graphs[3]. From this, we can show that, given an equivalence class $[e]_E$, all the first ends of the edges of $[e]_E$ are equivalent, or, in other words, that they are all in the same equivalence class. Then, given an edge e in E_B, its equivalence class $[e]_E$ is isomorphic to the edge $([fst(e)]_V, [snd(e)]_V)$. The morphism nature of π is based on the same kind of arguments.

3.4 Co-universal construction

If we consider the definitions of the coproduct and the coequalizer, we may notice a common point between this constructions: they have some property P (having injection arrows/coequalizing f and g) and, if there is another construction which have the same property P, there exists a unique arrow from the coproduct/coequalizer to this other construction. This fact is captured by the notion of co-universal construction. To define this notion, we first define the notion of initial object.

Definition 4 (Initial object)**.** Given a category, an initial object 0 of this category is an object such that, for every object B of this category, there exists a unique arrow from 0 to B (see figure 10).

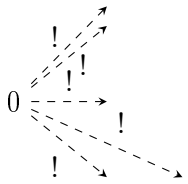


Figure 10: An initial object. There exists a unique arrow from this object to every other object of the category.

[3] As stated through the equations $f_V(fst(e)) = fst(f_E(e))$ and $f_V(snd(e)) = snd(f_E(e))$.

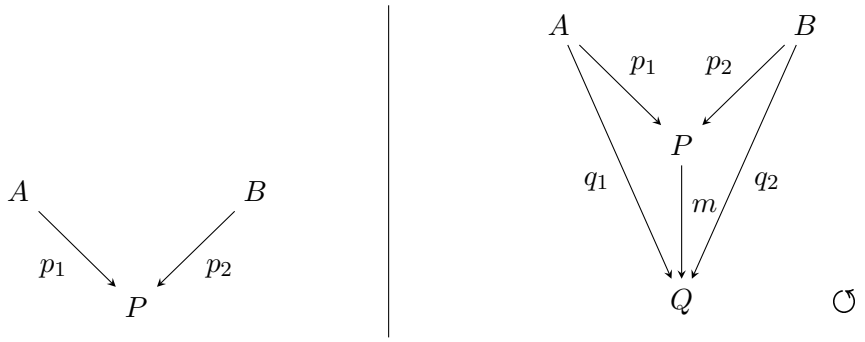


Figure 11: A triple (P, p_1, p_2) and a morphism m of the category $\mathcal{C}_{AB}$.

Remark. The initial object of a category is unique up to isomorphism.

Example. In the category **Set**, the initial object is the empty set. Given any set B, there is exactly one arrow from the empty set to B: the empty function, whom domain is empty.

Example. Similarly, the initial object of the category **Graph** is the empty graph, and the unique arrows from it to any other object are the empty morphisms, whom components are empty functions.

Definition 5 (Co-universal construction)**.** Given a category $\mathcal{C}$ and a property P, we consider the category $\mathcal{C}_P$ of all constructions of the category $\mathcal{C}$ satisfying P. An initial object of this category is called a co-universal construction.

Remarks.
- As the initial object is unique up to isomorphism, so are the co-universal constructions.
- The constructions defined by a co-universal construction are said co-universal among constructions satisfying the property P, or to have the co-universality property.

Example (Coproduct)**.** This example is pictured on figure 11. In a category $\mathcal{C}$, given A and B, if we consider the triples (P, p_1, p_2) where $p_1 : A \to P$ and $p_2 : B \to P$, they form the objects of a category $\mathcal{C}_{AB}$. An arrow $m : (P, p_1, p_2) \to (Q, q_1, q_2)$ of $\mathcal{C}_{AB}$ is an arrow of $\mathcal{C}$ such that $q_1 \circ m = p_1$ and $q_2 \circ m = p_2$. An initial object of this category is a coproduct of A and B. Thus, the coproduct has the co-universality property.

3.5 Pushout

As said before, the intuition behind pushouts is that they put structures together without adding nor deleting anything [20].

Definition 6. A pushout of a pair of arrows $f : A \to B$ and $g : A \to C$ is an object P and a pair of arrows $g' : B \to P$ and $f' : C \to P$ s.t. $g' \circ f = f' \circ g$ and if $i : B \to X$ and $j : C \to X$ are such that $i \circ f = j \circ g$ then there is a unique $k : P \to X$ s.t. $i = k \circ g'$ and $j = k \circ f'$ (see figure 12).

Remark. Pushouts have the universality property, hence they are unique up to isomorphism.

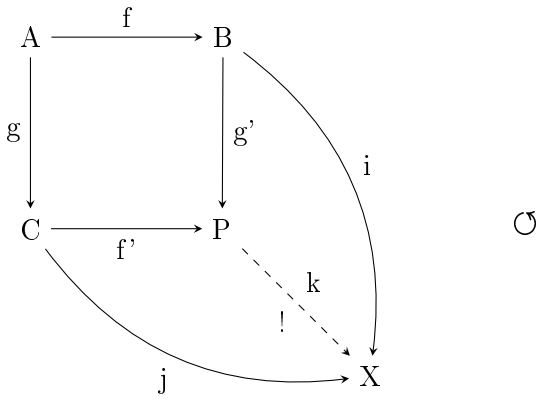


Figure 12: (P, f', g') is a pushout of f and g.

Example (In the category **Graph**)**.** An example of pushout in the category **Graph** is pictured on the figure 13. As we can see on this figure, the graph D, part, with f' and g', of the pushout of f and g, corresponds to a merge of B and C with respect to A. Indeed, the elements of A and their image through f and g serve as a marker of common elements of B and C, i.e. the elements identified as identical and which must be present only once in the merge. Moreover, f' and g' give us information on the origin of each element of D. In fact, if an element of D is in the image of f' and g', we know that it is the result of the merge of its antecedents, otherwise we can find from which graph it comes from.

As the pushout is an interesting operator, we need a constructive way to obtain it if we want to use it in a tool. Fortunately, such a constructive definition exists, and is given by the following theorem.

Theorem 1. *In a category where every two objects have a coproduct, and every two arrows with common domain and codomain have a coequalizer, every two arrow $f : A \to B$ and $g : A \to C$ with common domain A have a pushout. Moreover, this pushout can be constructed by taking the coproduct of the codomains B and C of f and g, then considering the coequalizer $h : B \uplus C \to D$ of $\iota_B \circ f$ and $\iota_C \circ g$, the triple $(D, h \circ \iota_C, h \circ \iota_B)$ is a pushout of f and g (see figure 14).*

4 The category Model

4.1 A category of objects and mappings

The various examples of model composition introduced in section 2.2 have in common that they rely on mappings to express links between models. Thus, it appears quite natural to define a category with models as objects and mappings (i.e. relations between models) as arrows.

A model is a set of classes linked through references. Though, for reasons which will be made clearer in section 5.2, we need to be able to manipulate references between models as first class objects, and not as simple links. Thus, we introduce the notion of edge, which link references and classes. Restrictions are made on the edges, to ensure that a reference may be

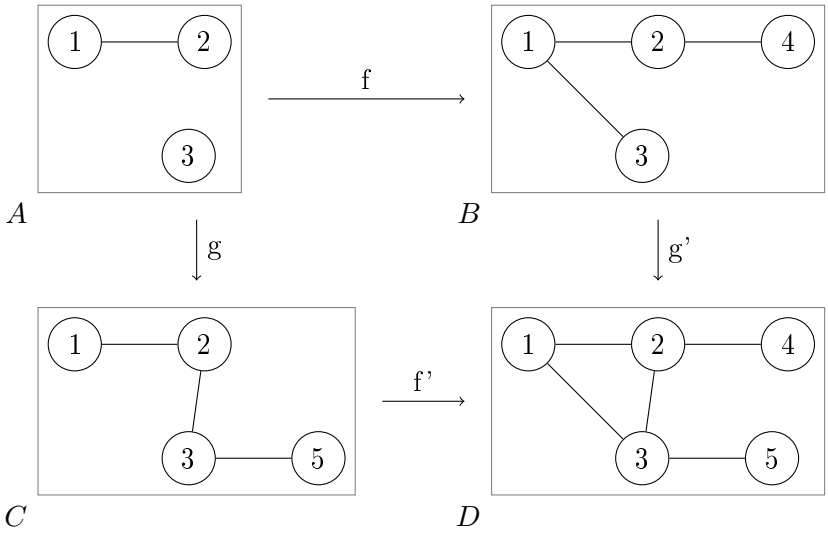


Figure 13: A pushout in the category **Graph**.

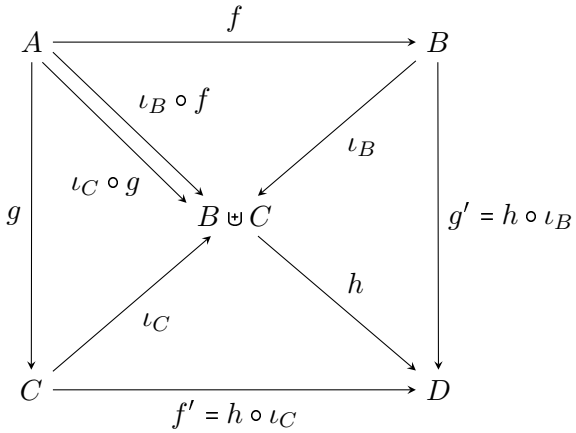


Figure 14: A constructive definition of the pushout.

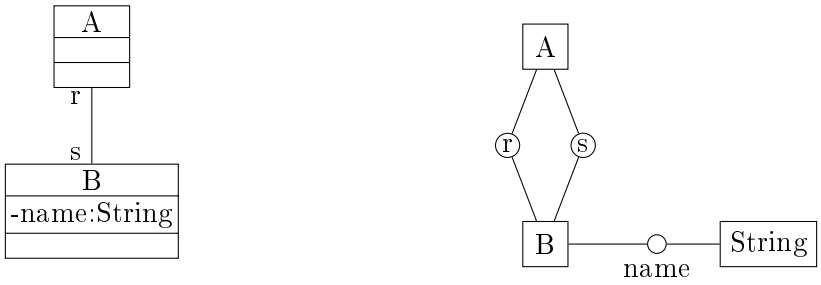


Figure 15: An example of model and its formal representation. Orientation of the edges is not represented.

linked to at most two objects, as if it was a link. Nevertheless it can be linked to only one, or even to no class, which would not have been easy to manage if they were simple links. Finally, as references are directed, the information on the direction of a reference is put in its edges.

Definition 7 (Model)**.** A model is a set C of classes, a set R of references and a set E of edges such that

1. $E \subseteq C \times R \cup R \times C$
2. $\forall (r, c) \in E \cap R \times C, \forall c' \in C, (r, c') \in E \iff c = c'$
3. $\forall (c, r) \in E \cap C \times R; \forall c' \in C, (c', r) \in E \iff c = c'$

Remark. Given an edge e, we use the notations $e^{(c)}$ and $e^{(r)}$ to represent the class and reference ends of e.

As stated before, the information on the direction of a reference is stored in its edges: if $(c, r) \in E$, r is a reference *from* class c, whereas if $(r, c) \in E$, r is a reference *to* c. The points 2 and 3 say that a reference is from at most one class and to at most one class.

An example of usual model and its formal representation are given on figure 15.

A model mapping is given by three relations: a relation on the classes, a relation on the references and a relation on the edges, with an additional constraint to ensure that these mappings preserve the structure of the models.

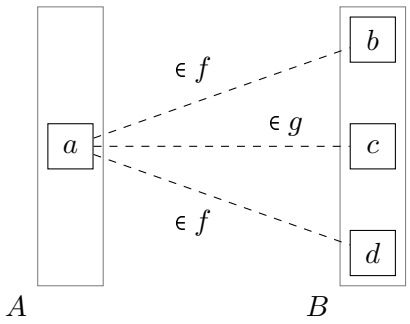
Definition 8 (Model mapping)**.** Given two models $A = (C_A, R_A, E_A)$ and $B = (C_B, R_B, E_B)$, a model mapping M between A and B is a triple (M_C, M_R, M_E) such that $M_C \subseteq C_A \times C_B$, $M_R \subseteq R_A \times R_B$, $M_E \subseteq E_A \times E_B$ and $\forall e_a, e_b \in M_E, (e_a^{(c)}, e_b^{(c)}) \in M_C$ and $(e_a^{(r)}, e_b^{(r)}) \in M_R$.

An element (a, b) of $M_C \cup M_R \cup M_E$ is called an alignment rule.

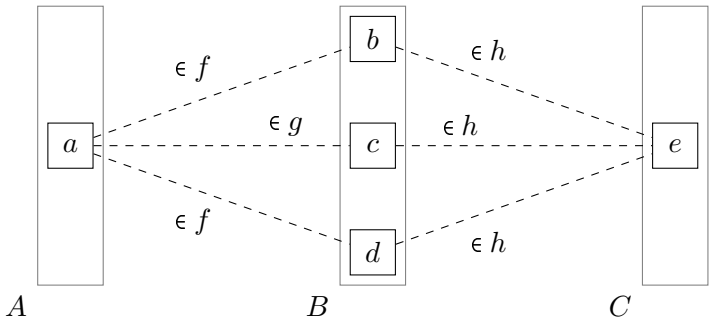
For the sake of simplicity, we will use *mapping* instead of *model mapping*.

The category **ModelRel**, with models as objects and model mappings as arrows, has been properly defined, as well as its coequalizers, coproducts and pushouts. Furthermore, the category and all these constructions have been implemented using OCaml[4]. When we tried to use them, a fact appeared: the coequalizer did not behave really as expected (see figure 16), though the pushout did. After a little investigation, it appeared that it was because when we

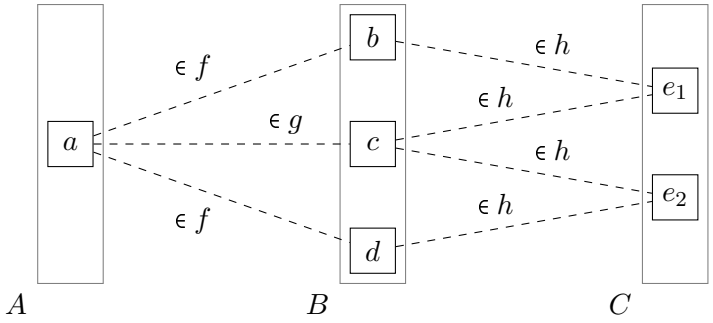
[4] `http://caml.inria.fr/ocaml/index.fr.html`



The two arrows $f, g : A \to B$



The expected coequalizer of f and g.



The real coequalizer of f and g.

Figure 16: An example where the coequalizer does not behave as expected in the category **ModelRel**. In the top, two relation $f, g : A \to B$. In the middle, the expected coequalizer of these two arrows. In the bottom, the real coequalizer, which has the co-universality property.

used the pushouts, we used as arrows only partial morphisms. Partial morphism are a special kind of relations, with which the coequalizer, and thus the pushout, behaves as expected. To understand how this happen, see section 5.1. So we simplified our category to use partial morphisms as arrows instead of mappings, this lead to the definition of the category **Model**.

4.2 The category Model

The category **Model** has as objects models and as arrows partial model morphisms. The models have already been defined in section 4.1, but we still have to define the partial model morphisms. We define them as three partial functions on classes, references and edges, and add some restriction to ensure that they preserve the model structure. Their partial nature will be very useful when doing pushouts, as explained in section 4.4. We begin by defining what is a partial function.

Definition 9 (Partial function)**.** Given two sets A and B, a partial function $f : A \to B$ is a function $f : A' \to B$ with A' a subset of A. A is called domain of f, noted $\mathrm{dom}(f)$, while A' is called domain of definition of f, noted $\mathrm{def}(f)$. For $x \in \mathrm{dom}(f) \backslash \mathrm{def}(f)$, $f(x)$ is undefined.

The composition of partial functions is the classical relation composition, i.e. $x \in \mathrm{def}(g \circ f) \iff (x \in \mathrm{def}(f)$ and $f(x) \in \mathrm{def}(g))$ and $g \circ f(x) = g(f(x))$.

From a partial function $f : A \to B$, we can turn it into a total function by adding a special value $\perp$, meaning undefined, to B, yielding $B^\perp = B \cup \{\perp\}$ and defining $f^\perp : A \to B^\perp$ by

$$f^\perp(x) = \begin{cases} f(x) & \text{if } x \in \mathrm{def}(f), \\ \perp & \text{either.} \end{cases}$$

The function $f^\perp$ is called the total function induced by f. Then, when we want to compose such induced total functions, we have to extend the second function to take into account the new value $\perp$. So we consider that $g^\perp(\perp) = \perp$. This view of partial functions, strictly equivalent to their definition, will be very helpful in later proofs.

Definition 10 (Partial model morphism)**.** Given two models $A = \{C_A, R_A, E_A\}$ and $B = \{C_B, R_B, E_B\}$, a partial model morphism $f : A \to B$ is a triple (f_C, f_R, f_E) such that $f_C : C_A \to C_B$, $f_R : R_A \to R_B$ and $f_E : E_A \to E_B$ are partial functions such that

$$\forall e \in \mathrm{def}(f_E), e^{(c)} \in \mathrm{def}(f_C), e^{(r)} \in \mathrm{def}(f_R), f_C(e^{(c)}) = (f_E(e))^{(c)} \text{ and } f_R(e^{(r)}) = (f_E(e))^{(r)} \tag{1}$$

The goal of the constraint (1) is to preserve the structure of models: if an edge is mapped with another edge, then their ends are mapped accordingly.

Remark. For the sake of simplicity, we may use *partial morphism*, *model morphism* or simply *morphism* instead of *partial model morphism*.

To define a category, we need to be able to compose arrows and the identity arrows. We thus formally define partial model morphisms composition and the identity model morphism.

Definition 11 (Partial model morphisms composition)**.** The composition of two partial model morphisms is done component-wise, i.e. $g \circ f = (g_C \circ f_C, g_R \circ f_R, g_E \circ f_E)$.

Proof. We need to check that the composed morphism satisfies the condition (1).
Let $e \in \text{def}(g_E \circ f_E)$ be an edge. By definition of the partial function composition, $e \in \text{def}(f_E)$ and $f_E(e) \in \text{def}(g_E)$. As f and g are morphisms, we have that $e^{(c)} \in \text{def}(f_C)$, $(f_E(e))^{(c)} \in \text{def}(g_C)$, and that $f_C(e^{(c)}) = (f_E(e))^{(c)}$ and $g_C((f_E(e))^{(c)}) = (g_E(f_E(e)))^{(c)}$. So $g_C \circ f_C(e^{(c)}) = (g_E \circ f_E(e))^{(c)}$.
The arguments are the same for $e^{(r)}$. □

Definition 12 (Identity model morphism)**.** Given a model A, the identity morphism of A is a model morphism $id_A : A \to A$ such that $(id_A)_C$,$(id_A)_R$ and $(id_A)_E$ are the identity functions on C_A, R_A and E_A respectively.

Remark. id_A is a total model morphism, a special case of partial model morphism where the tree components are total functions.

Proof. We need to check that id_A satisfies the condition (1).
Let $e \in E_A$ be an edge. As $(id_A)_C$ is the identity function on C_A, $(id_A)_C(e^{(c)}) = e^{(c)}$. As $(id_A)_E$ is the identity function on E_A, $(id_A)_E(e) = e$. So $((id_A)_E(e))^{(c)} = e^{(c)} = (id_A)_C(e^{(c)})$.
The same arguments hold for $e^{(r)}$. □

Definition 13 (**Model** category)**.** The category **Model** has models as objects and partial model morphisms as arrows. Composition of arrows is the composition of morphisms. Identity arrows are identity morphisms.

Proof. We have to check if the four properties of definition 1 are satisfied.

- Every morphism has obviously a source and a target
- Given two morphisms $f : A \to B$ and $g : B \to C$, the composed morphism $g \circ f : A \to C$ always exists, even if it sometimes an empty function ($\text{def}(g \circ f) = \varnothing$).
- The composition of partial functions is associative and the composition of morphisms is done component-wise, so composition of morphisms is associative:

$$\begin{aligned} h \circ (g \circ f) &= (h_C \circ (g_C \circ f_C), h_R \circ (g_R \circ f_R), h_E \circ (g_E \circ f_E)) \\ &= ((h_C \circ g_C) \circ f_C, (h_R \circ g_R) \circ f_R, (h_E \circ g_E) \circ f_E) \\ &= (h \circ g) \circ f \end{aligned}$$

- The identity functions are neutral for partial function composition and the composition of morphisms is done component-wise, so the identity morphism is neutral for morphism composition. □

4.3 Coequalizer and coproduct

To be able to construct the pushout, we need the coequalizer and the coproduct.

Definition 14 (Coequalizer)**.** Let $f, g : A \to B$ be two arrows of the category **Model**. The coequalizer $h : B \to D$ of f and g is defined by the following construction.
Consider $\equiv_C$ the smallest equivalence relation s.t. $\forall a \in C_A, f_C^{\perp}(a) \equiv_C g_C^{\perp}(a)$. Let $C_D^{\perp}$ be the set of equivalence classes of $\equiv_C$ and $h_C^{\perp}$ be the canonical projection function. Then C_D is obtained by removing the equivalence class of $\perp$ and h_C is the partial function induced by $h_C^{\perp}$.

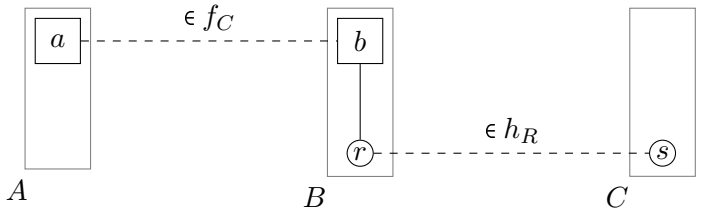


Figure 17: h is the coequalizer of f and g. f and h are represented on the figure and g is the empty function.

The same construction holds for R_D and h_R.
The construction of E_D is a bit different. Like previously we construct $E_D^\perp$, but we add a constraint on the ends: we keep an equivalence class W in $E_D^\perp$ if and only if $\{w^{(c)}|w \in W\} \in C_D$[5] and $\{w^{(r)}|w \in W\} \in R_D$, otherwise it is merged with the equivalence class of $\perp$. Then, for $e \in E_D$, we have that $e^{(c)} = h_C(e_b^{(c)})$ and $e^{(c)} = h_R(e_b^{(r)})$ for some $e_b \in E_B$ such that $h_E(e_b) = e$.

Remark. The constraint on the ends of the edges is there to avoid any dangling edge. An example where this constraint is not met is given in figure 17. In this example, if the edge of the model B was added in model C, it would have been dangling, as b does not appear in this model.

Proof. The proof is in four steps.

1. We check that D is well-defined, i.e. that for $e \in C_E$, $e^{(c)}$ and $e^{(r)}$ are well defined.
   Let $e \in E_D$ be an edge. By construction, $W = h_E^{-1}(e) = \{w \in E_B|h_E(w) = e\}$ is an equivalence class for $\equiv_E$. By the constraint on the ends, $W^{(c)} = \{w^{(c)}|w \in W\} \in C_D$. Thus $\forall e_b \in W$, $h_C(e_b^{(c)}) = W^{(c)}$, hence $e^{(c)} = h_C(e_b^{(c)})$ for some $e_b \in W$ does not depend of the $e_b^{(c)}$ chosen.
   The same arguments hold for $e^{(r)}$.

2. We check that h is a valid mapping, i.e. that the property (1) holds.
   Consider $e_b \in \text{def}(h_E)$ and $e = h_E(e_b)$. Obviously, $(h_E(e_b))^{(c)} = e^{(c)}$. By the constraint on the edges, we have that $e_b^{(c)} \in \text{def}(h_C)$. As seen in the first point of this proof, $e^{(c)} = h_C(e_b^{(c)})$, hence $(h_E(e_b))^{(c)} = h_c(e_b^{(c)})$. As the same arguments hold for $e_b^{(r)}$, the constraint (1) is satisfied.

3. We check that h *do* coequalize f and g, i.e. $h \circ f = h \circ g$.
   Let $a \in C_A$ be a class of A and $b = f_C^\perp(a)$ its image by $f_C^\perp$. By construction, $b \in h_C^\perp(b)$ the equivalence class of b for $\equiv_C$. By definition of $\equiv_C$, $f_C^\perp(a) \equiv_C g_C^\perp(a)$, so $g_C^\perp(a) \in h_C^\perp(b)$, and $h_C^\perp(g_C^\perp(a)) = h_C^\perp(b) = h_C^\perp(f_C^\perp(a))$. Hence $h_C \circ f_C = h_C \circ g_C$.
   As the same arguments hold for h_R and h_E, we have that $h \circ f = h \circ g$.

4. We check that D satisfies the universal property.
   Let D' be a model and $h' : B \to D'$ such that $h' \circ f = h' \circ g$. As $\equiv_C$ is the minimal equivalence relation such that $f_C^\perp(a) = g_C^\perp(a)$, $(h'_C)^\perp$ has to be constant on any equivalence

[5] As $E_D^\perp$ is the set of equivalence class of E_B for $\equiv_E$, we have that $E_D^\perp \subseteq \mathcal{P}(E_B)$, so $W \in \mathcal{P}(E_B)$, hence $\{w^{(c)}|w \in W\} \in \mathcal{P}(C_B)$ and we are allowed to write that $\{w^{(c)}|w \in W\} \in C_D \subseteq \mathcal{P}(C_B)$.

class of $\equiv_C$ (otherwise it would yield a finer grain equivalence relation). So we can define $k_C : C_D \to C_{D'}$ by $k_C(h_C(b)) = h'_C(b)$.
We use the same procedure to define k_R and k_E.
The uniqueness comes from the minimality.

□

Definition 15 (Coproduct)**.** The coproduct of two models A and B is given by $A \uplus B = (C_A \uplus C_B, R_A \uplus R_B, E_A \uplus E_B)$ together with the canonical injection arrows $\iota_A = (\iota_{C_A}, \iota_{R_A}, \iota_{E_A})$ and $\iota_B = (\iota_{C_B}, \iota_{R_B}, \iota_{E_B})$.

Proof. Consider D a model together with $f : A \to D$ and $g : B \to D$. As $C_A \uplus C_B$ is the coproduct of C_A and C_B in the category **Set**, we have easily the existence and uniqueness of $[f, g]_C : C_A \uplus C_B \to C_D$. We have as well $[f, g]_R$ and $[f, g]_E$. By their commutativity property, we have immediately that $[f, g]$ is a model morphism. Its uniqueness comes directly from the uniqueness of its components. □

4.4 Pushout

As stated in theorem 1, the pushout of two arrows $f : A \to B$ and $g : A \to C$ is easily obtained by taking the coproduct $B \uplus C$ and then the coequalizer of $\iota_B \circ f$ and $\iota_C \circ g$. As explained in section 3.5, a good intuition is that the pushout consists in putting things together, with nothing left and nothing added. However, if this intuition is good with total functions, it become partially false when considering partial functions, which is our case here. Indeed, when taking the coequalizer of two arrows, some elements of their codomain, namely those of the same equivalence class that the undefined value, are not mapped in the coequalizer, whereas the others are injected in their equivalence classes. Thus, the latter ones are preserved whereas the former ones are, in some sense, deleted.

As we use the coequalizer to construct the pushout, we have as well some elements which disappear. In figure 18, the class a of model C is the image of the one of A by g, though the latter has no image by f. So, when we apply the coequalizer, a of C is in the equivalence class of $\perp$ and is not mapped in D, resulting in its deletion.

5 Defining model composition through the category Model

As explained in the introduction, the goal of this work was to find a unified formal framework to describe various operators of model composition. In this section, we will see how we can use the pushout in the category **Model** to formalize different kinds of composition.

5.1 Merge

Following the intuition of the pushout, the most obvious way to compose models using pushout is the merge. Given B and C two models and M a mapping between them (as defined in section 4.1), we want to merge them w.r.t. M. To do this, we first transform M in a model A together with two morphisms f and g, using the following procedure. For every alignment rule $(c_b, c_c) \in M_C$, we add an element c_a to C_A and define $f_C(c_a) = c_b$ and $g_C(c_b) = c_c$. We use the same procedure for M_R and M_E. We can easily prove that f and g are model morphisms,

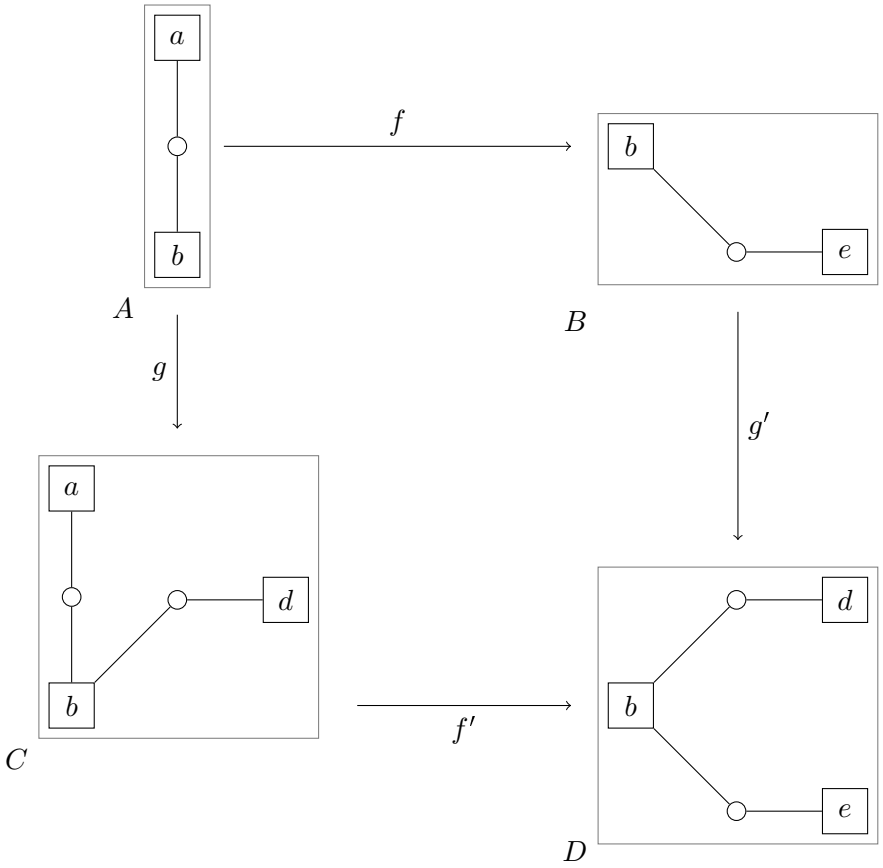


Figure 18: The pushout of f and g merges B and C w.r.t. A, creating a combination of the two models where b is linked with d and e, while a is deleted.

using the fact that M is a model mapping. Indeed, the structure preserving property of M immediately leads to the structure preserving property of f and g.

Now that we have two arrows with common domain, we can compute their pushout in **Model**. This pushout D is the merge of B and C along M. Indeed, each element a of A represent an alignment rule, and its images by f and g are the ends of this alignment rule. Thus, according to the construction of the pushout, every couple of elements mapped by M will be merged, and the other ones will be added. Moreover, as f and g are obviously total morphisms, no element of B or C will be deleted, which is the behaviour expected from a merge, as specified for example in the MOF specification [15].

As an example, consider the figure 19, where two models B and C and a mapping M between them are pictured. Notice that M_C is a full relation and not a function, as a of A is linked to two classes in B. After the transformation of M into a model, we obtain the figure 20. Notice that f_C and g_C are morphisms. Finally, we do the pushout of f and g, and get the merge of B and C, as pictured on figure 21.

In fact, the pushout gives us more than just the merge of B and C, it also gives the morphisms f' and g'. These morphisms may reveal themselves really useful, as they enable to track from which elements a merged element comes from. As an example, in figure 21, we may think that a' has been deleted during the pushout. Though, if we examine g', we see that a' has an image by g' (which would not be the case if it was deleted) and that its image is a, so it has been merged with other elements in this element. Examining now the inverse image of a by f' and g', we find out that this element is a merge of a in C, a in B and a' in D.

5.2 Transformation

Another example of model composition is the transformation. A transformation is given by two models A and B and a morphism $f : A \to B$, A being the pattern to match, B giving what the transformed pattern is, and f giving the relation between the elements of A and those of B. Elements not in the domain of definition of f are deleted, those not in the image of f are added, while elements in the domain of definition of f are preserved in some sense (indeed, two elements of A will be merged together if they have the same image). This representation is close from the formalism of the single pushout approach for graph transformation [18].

As an example, consider the transformation rule pictured in figure 22. f maps elements of A on the element of the same name in C. Here, the deleted elements are a, b, the reference between a and b and its edges, and the edge between b and r. The created elements are c and the edge between c and r.

Given a transformation (A,B,f), we want to apply it to a model C. To do it, we give a total morphism $g : A \to C$ showing how C matches the pattern A. This morphism must be total, otherwise we would only match a part of A and not the whole pattern, applying the transformation where we should not. Then, if we compute the pushout D of f and g, we apply the transformation (A,B,f) on C. As an example, consider the figure 23 where we apply the previous transformation to some model C.

This example shows why we have references as first class citizens in our category: it allow us to specify that we want to keep the reference r. If references were modeled as simple links between classes, the link between c and e could have been added, but it would not have been the same link than the one between b and e.

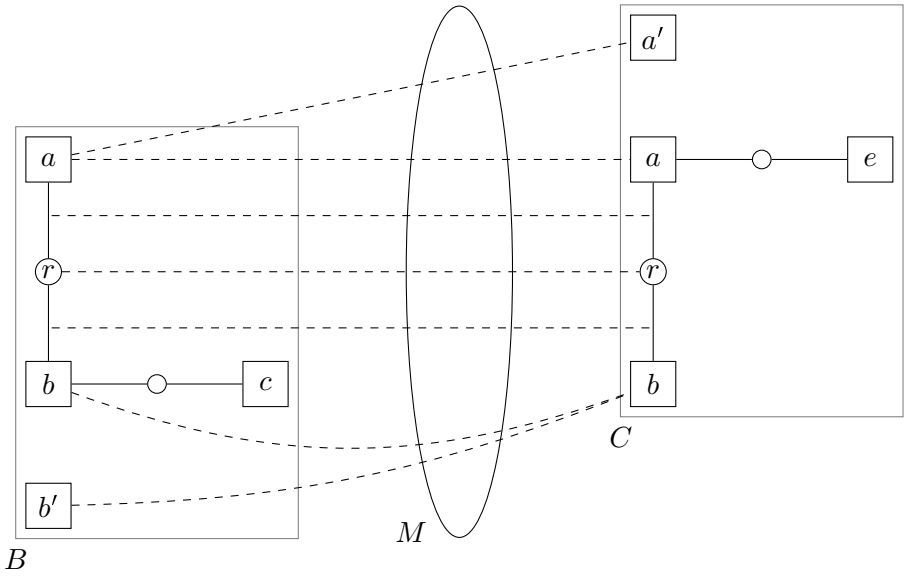


Figure 19: Two models B and C and a mapping M between them, represented in dashed lines

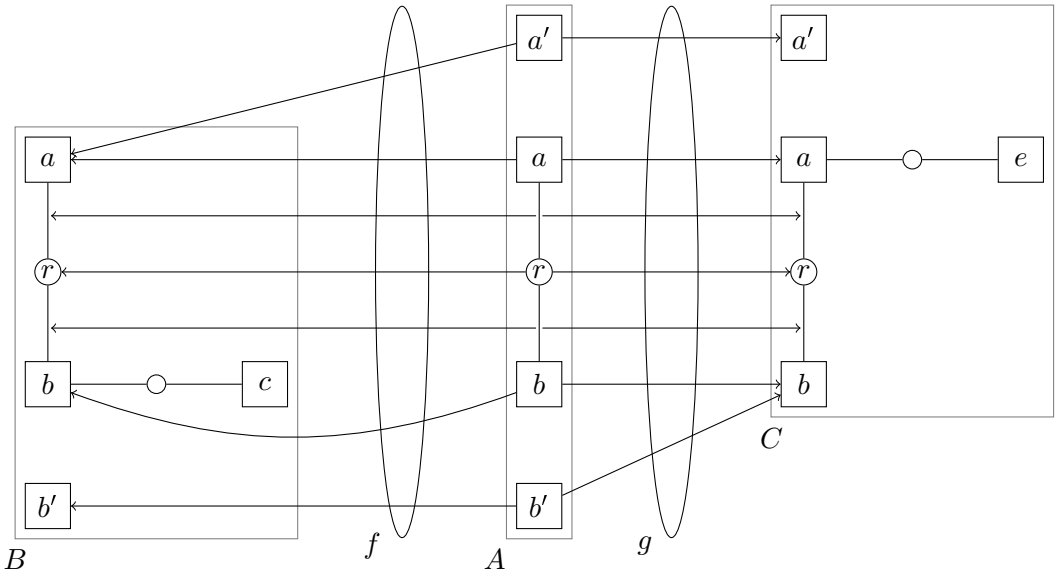


Figure 20: Two models B and C mapped by a third model A and two morphisms $f : A \to B$ and $g : A \to C$

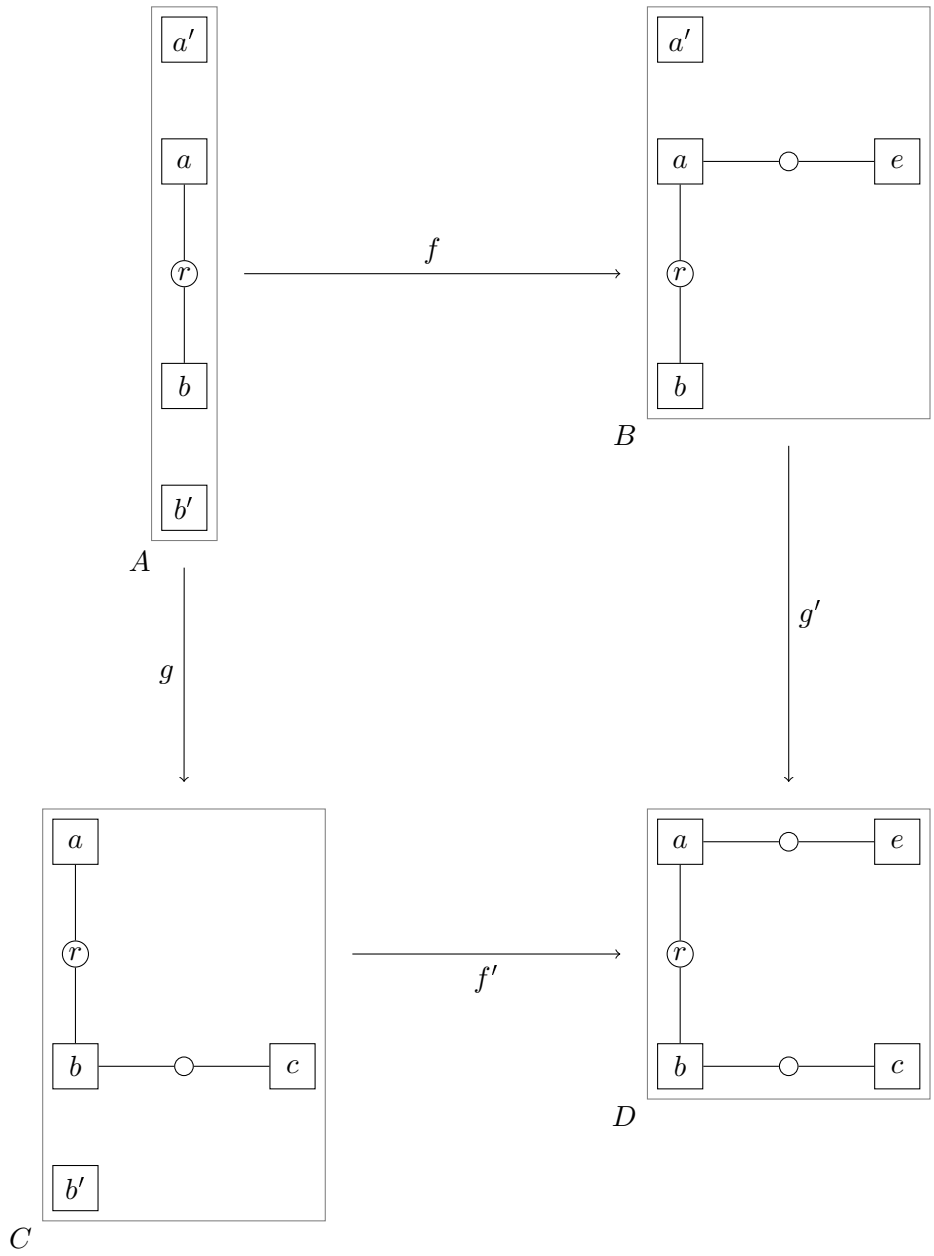


Figure 21: D is the merge of B and C with respect to A.

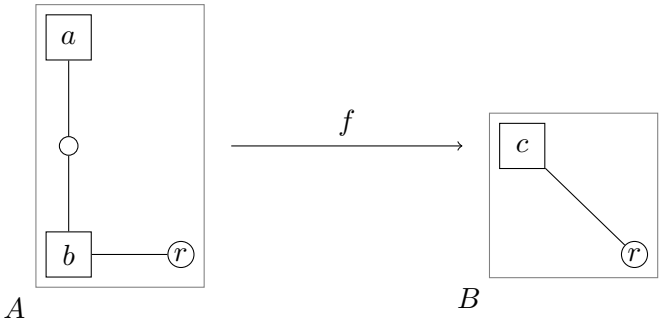


Figure 22: A transformation rule example.

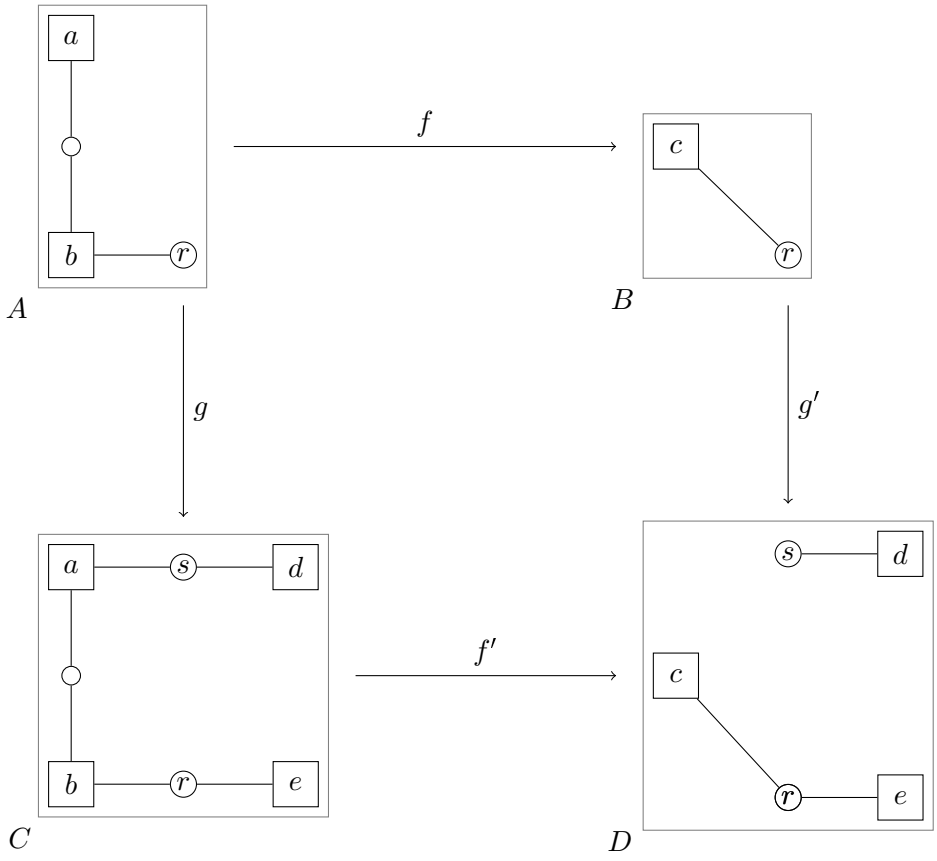


Figure 23: Application of the transformation (A, B, f) on the model C using the pushout. g is the match of A in C.

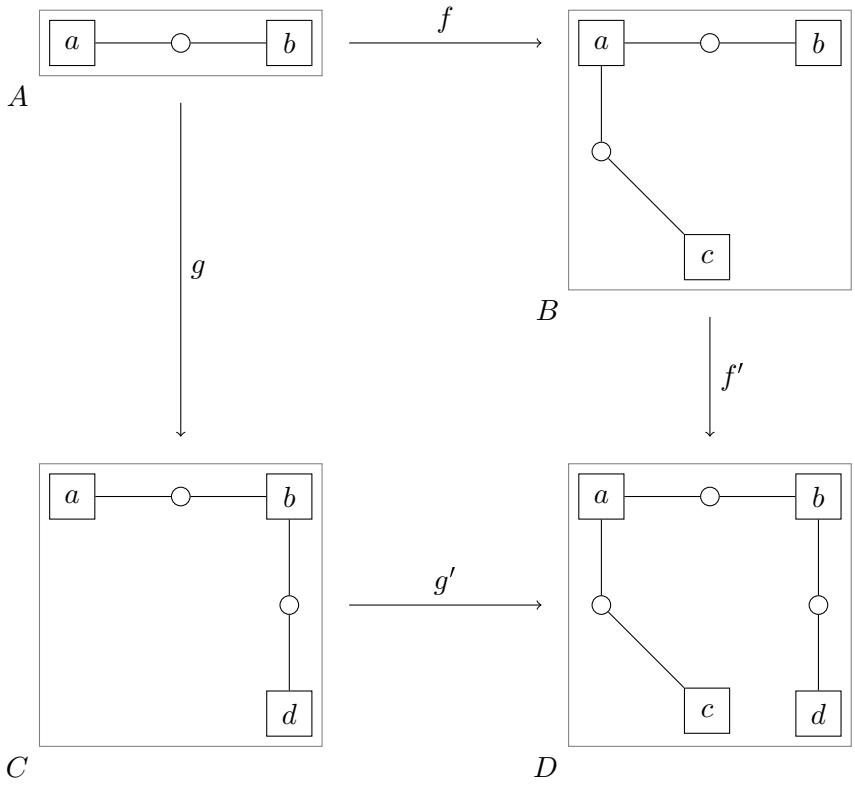


Figure 24: Given the pointcut A, we apply the advice B on the base model C by constructing the pushout D

5.3 Aspect weaving

An aspect is given by a pointcut A, which is the pattern to match, and an advice B, which shows what we add to this pointcut. To identify the elements of the pointcut A in B, we have in addition a morphism f between A and B, which must be both total and component-wise injective, as we do not want to lose any information by applying the advice. Then, to weave the aspect (A,B,f) in a base model C, given a total match g between A and C, we compute the pushout D of f and g. As an example consider figure 24.

6 Conclusion

We successfully introduced a unified framework for model composition using category theory. This framework, as expected, allows us to compare different compositions. For example, we can see that the difference between an aspect weaving and a merge is that in the former case, both f and g are injective. It is coherent with the idea that weaving an aspect should not suppress nor merge elements of the base model but only add elements, whereas in the case of model merge, elements may be merged together. In the same idea, we may notice that in the cases of merge and aspect weaving, the two morphisms are total, whereas it is not the case in transformation. It shows that transformation allows the deletion of elements, whereas the two other do not enable any information loss.

There is somehow still work to do on two aspects: the formalization of other compositions,

and the implementation in a tool. For the formalization of compositions, we are about to integrate model diff, which does the difference of two models, in versioning purpose for example. We have also ideas to integrate synchronization, using inspiration from what have been done with triple graph grammars [7]. The framework will be implemented in Kermeta[6], a metamodeling environment allowing to add behavior to models. On another point, we plan to take advantage of the high level of abstraction of category theory to prove the associativity and commutativity of model composition under some assumptions, without treating any specific composition.

References

[1] P.A. Bernstein, A.Y. Halevy, and R.A. Pottinger. A vision for management of complex models. *ACM Sigmod Record*, 29(4):55–63, 2000.

[2] Enrico Biermann, Claudia Ermel, Leen Lambers, Ulrike Prange, Olga Runge, and Gabriele Taentzer. Introduction to agg and emf tiger by modeling a conference scheduling system. *STTT*, 12(3-4):245–261, 2010.

[3] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12. ACM, 2006.

[4] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information preserving bidirectional model transformations. In Matthew B. Dwyer and Antónia Lopes, editors, *FASE*, volume 4422 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2007.

[5] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. *Fundam. Inform.*, 74(1):31–61, 2006.

[6] S. Eilenberg and S. Mac Lane. General theory of natural equivalences. *Trans. Amer. Math. Soc*, 58(2):231, 1945.

[7] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8:21–43, 2009. 10.1007/s10270-008-0089-9.

[8] Joseph A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[9] Object M. Group. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Technical report, November 2007.

[10] Object Management Group. Unified modeling language 2.1.2 super-structure specification. Specification Version 2.1.2, Object Management Group, November 2007.

[11] J.M. Jézéquel. Model driven design and aspect weaving. *Software and Systems Modeling*, 7(2):209–218, 2008.

[6] http://www.kermeta.org/

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, chapter 10, pages 220–242. Springer-Verlag, Berlin/Heidelberg, 1997.

[13] Saunders Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer-Verlag, 1971.

[14] ML Minsky. Matter, Mind and Models Semantic Information Processing, 1968.

[15] Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[16] Benjamin C. Pierce. *Basic category theory for computer scientists.* Foundations of computing. MIT Press, 1991.

[17] R.A. Pottinger and P.A. Bernstein. Merging models based on given correspondences. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, page 873. VLDB Endowment, 2003.

[18] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[19] Mehrdad Sabetzadeh and Steve M. Easterbrook. Analysis of inconsistency in graph-based viewpoints: A category-theoretic approach. In *ASE*, pages 12–21. IEEE Computer Society, 2003.

[20] Mehrdad Sabetzadeh and Steve M. Easterbrook. An algebraic framework for merging incomplete and inconsistent views. In *RE*, pages 306–318. IEEE Computer Society, 2005.

[21] Mehrdad Sabetzadeh, Shiva Nejati, Steve M. Easterbrook, and Marsha Chechik. Global consistency checking of distributed models with tremer+. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 815–818. ACM, 2008.