

A Category-Theoretical Approach to the Formalisation of Version Control in MDE

Adrian Rutle¹, Alessandro Rossini², Yngve Lamo¹, and Uwe Wolter²

¹ Bergen University College, P.O. Box 7030, 5020 Bergen, Norway
{aru,yla}@hib.no

² University of Bergen, P.O. Box 7803, 5020 Bergen, Norway
{rossini,wolter}@ii.uib.no

Abstract. In Model-Driven Engineering (MDE) models are the primary artefacts of the software development process. Similar to other software artefacts, models undergo a complex evolution during their life cycles. Version control is one of the key techniques which enables developers to tackle this complexity. Traditional version control systems are based on the *copy-modify-merge* paradigm which is not fully exploited in MDE because of the lack of model-specific techniques. In this paper we give a formalisation of the copy-modify-merge paradigm in MDE. In particular, we analyse how common models and merge models can be defined by means of category-theoretical constructions. Moreover, we show how the properties of those constructions can be used to identify model differences and conflicting modifications.

1 Introduction and Motivation

Since the beginning of computer science, raising the abstraction level of software systems has been a continuous process. One of the latest steps in this direction has led to the usage of modelling languages in software development processes. Software models are indeed abstract representations of software systems which are used to tackle the complexity of present-day software by enabling developers to reason at a higher level of abstraction. In Model-Driven Engineering (MDE) models are first-class entities of the software development process and undergo a complex evolution during their life-cycles. As a consequence, the need for techniques and tools to support model evolution activities such as version control is increasingly growing.

Present-day MDE tools offer a limited support for version control of models. Typically, the problem is addressed using a *lock-modify-unlock* paradigm, where a repository allows only one developer to work on an artefact at a time. This approach is workable if the developers know who is planning to do what at any given time and can communicate with each other quickly. However, if the development group becomes too large or too spread out, dealing with locking issues might become a hassle.

On the contrary, traditional version control systems such as Subversion enable efficient concurrent development of source code. These systems are based on the *copy-modify-merge* paradigm. In this approach each developer accesses a repository and creates a local working copy – a snapshot of the repository’s files and directories. Then, the developers modify their local copies simultaneously and independently. Finally, the local modifications are merged into the repository. The version control system assists with

the merging by detecting conflicting changes. When a conflict is detected, the system requires manual intervention of the developer.

Unfortunately, traditional version control systems are focused on the management of text-based files, such as source code. That is, difference calculation, conflict detection, and source code merge are based on a per-line textual comparison. Since the structure of models is graph-based rather than text- or tree-based [1], the existing techniques are not suitable for MDE.

During the last years, research has lead to various outcomes related to model evolution: [2] for the difference calculation, [3] for the difference representation, [4] for the conflict detection, and [5] for syntactic software merging that exploits the graph-based structure(s) of programs, to cite a few. However, the proposed solutions are not formalised enough to enable automatic reasoning about model evolution. For example, operations such as *add*, *delete*, *rename* and *move* are given different semantics in different works/tools. In addition, concepts such as *synchronisation*, *commit* and *merge* are only defined semiformaly. Moreover, the terminology is not precise and unique, e.g. the terms “create”, “add” and “insert” are frequently used to refer to the same operations.

Our claim is that the adoption of the copy-modify-merge paradigm is necessary to enable effective version control in MDE. This adoption requires formal techniques which are targeting graph-based structures. The goal of this paper is the formalisation of the copy-modify-merge paradigm in MDE. In particular, we show that common models and merge models can be defined as pullback and pushout constructions, respectively. For our analysis we use the Diagram Predicate Framework (DPF)¹ [6,7,8] which provides a formal approach to modelling based on category theory – the mathematics of graph-based structures. In addition, DPF enables us to define a language to represent model differences and a logic to detect conflicting modifications.

The rest of the paper is structured as follow. Section 2 provides a brief introduction to DPF. Then Section 3 outlines a motivating example, and gives the formalisation of the concepts of version control. In Section 4 the state-of-the-art of research in version control is summarised. Finally, in Section 5 some concluding remarks and ideas for future work are given.

2 Diagram Predicate Framework

Diagram Predicate Framework (DPF), is a diagrammatic formalism for the definition and reasoning about modelling languages, (meta)models and model transformations. The formalism is based on category theory and first order logic; it combines the mathematical rigour – which is necessary to enable automatic reasoning – with the intuitiveness of diagrammatic notations [9]. DPF’s usage in the formalisation of concepts in (meta)modelling and model transformations are discussed in [8] and [7,10], respectively. This section includes only a short description of the basic concepts of DPF such as *signatures*, *constraints* and *diagrammatic specifications*.

In DPF, software models are represented by diagrammatic specifications². These diagrammatic specifications are structures which consist of a graph and a set of constraints.

¹ Formerly named Diagrammatic Predicate Logic (DPL).

² In this paper the terms *model* and *diagrammatic specification* are used interchangeably.

The graph represents the structure of the model. Predicates from a predefined diagrammatic signature are used to add constraints to the graph [6]. Each modelling language L corresponds to a diagrammatic signature Σ_L and a metamodel MM_L . L -models are represented by Σ_L -specifications where the underlying graphs are instances of the metamodel MM_L [8]. Signatures, constraints and diagrammatic specifications are defined as follows:

Definition 1 (Signature). A (diagrammatic predicate) signature $\Sigma := (\Pi, \alpha)$ is an abstract structure consisting of a collection of predicate symbols Π with a mapping that assigns an arity (graph) $\alpha(p)$ to each predicate symbol $p \in \Pi$.

Definition 2 (Constraint). A constraint (p, δ) in a graph G is given by a predicate symbol p and a graph homomorphism $\delta : \alpha(p) \rightarrow G$, where $\alpha(p)$ is the arity of p .

Definition 3 (Diagrammatic Specification). A Σ -specification $S := (G(S), S(\Pi))$, is given by a graph $G(S)$ and a set $S(\Pi)$ of constraints (p, δ) in $G(S)$ with $p \in \Pi$.

Table 1 shows a sample signature $\Sigma = (\Pi, \alpha)$ which consists of a collection of useful predicates such as [cover], [key] etc. The first column of the table shows the names

Table 1. A sample signature Σ

Π	α	Proposed visualisation	Intended semantics
[total]	$1 \xrightarrow{x} 2$		$\forall a \in A : f(a) \geq 1$
[key]	$1 \xrightarrow{x} 2$		$\forall a, a' \in A : a \neq a' \text{ implies } f(a) \neq f(a')$
[single-valued]	$1 \xrightarrow{x} 2$		$\forall a \in A : f(a) \leq 1$
[cover]	$1 \xrightarrow{x} 2$		$\forall b \in B : \exists a \in A \mid b \in f(a)$
[isA]	$1 \xrightarrow{x} 2$		$f = \phi$ where $\phi : B \rightarrow A$ is a persistent extension and $ \phi$ is its reduct.
[containment]	$1 \xrightarrow{x} 2$		$\forall b \in B, \exists a \in A \mid b \in f(a)$ and $\forall g : X \rightarrow B, \forall x \in X$ if $b \in g(x)$ then $f = g, X = A$ and $a = x$
[inverse]	$1 \begin{matrix} \xrightarrow{x} \\ \xleftarrow{y} \end{matrix} 2$		$\forall a \in A, \forall b \in B : b \in f(a) \text{ iff } a \in g(b)$
[jointly-key]	$1 \xrightarrow{x} 2$ $\downarrow y$ 3		$\forall a, a' \in A : a \neq a' \text{ implies } f(a) \neq f(a') \text{ or } g(a) \neq g(a')$

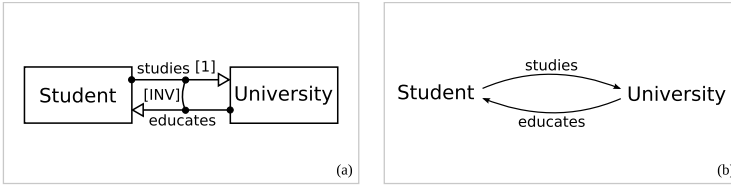


Fig. 1. A Diagrammatic Specification: (a) $S = (G(S), S(II))$, (b) its graph $G(S)$

of the predicates. The second and the third columns show the arities and a possible visualisation of these predicates, respectively. In the fourth column, the intended semantic of each predicate is specified. These predicates in Table 1 allow for specifying some useful properties and constraints that a modeller would define for a structural model. In addition, the signature can be extended with custom-defined predicates. Typically in structural models, model elements are interpreted as sets and arrows as multivalued functions $f : A \rightarrow \wp(B)$, i.e. an arrow without constraints stands for an arbitrary multivalued function. For example, in UML class diagrams the intended semantics of an association between two classes is that the instances of those two classes have a many-to-many relationship.

Fig. 1a shows an example of a Σ -specification $S = (G(S), S(II))$. S specifies the structural model of a simple information system for universities. $G(S)$ in Fig. 1b is the graph of S without any constraints on it. In S , every university educates *one or more* students; this is forced by the constraint ($[total], \delta_1$) on the arrow *educates* (see Table 2). Moreover, every student studies at *exactly one* university; this is forced by the constraint ($[single-valued], \delta_2$) on the arrow *studies*. Another property of S is that the functions *studies* and *educates* are inverse of each other, i.e. $\forall u \in University : u = studies(educates(u))$ and $\forall s \in Student : s \in educates(studies(s))$. This is forced by the constraint ($[inverse], \delta_4$) on *studies* and *educates*.

Table 2. Diagrams $(p, \delta) \in S(II)$

(p, δ)	$\alpha(p)$	$\delta(\alpha(p))$
$([total], \delta_1)$	$1 \xrightarrow{x} 2$	$University \xrightarrow{educates} Student$
$([single-valued], \delta_2)$	$1 \xrightarrow{x} 2$	$Student \xrightarrow{studies} University$
$([cover], \delta_3)$	$1 \xrightarrow{x} 2$	$University \xrightarrow{educates} Student$
$([inverse], \delta_4)$	$1 \begin{matrix} \xrightarrow{x} \\ \xleftarrow{y} \end{matrix} 2$	$Student \begin{matrix} \xrightarrow{studies} \\ \xleftarrow{educates} \end{matrix} University$

3 Version Control in MDE

The problem of version control in MDE is formalised in terms of category-theoretical constructs. It should be noted that our reasoning is applicable both at model and meta-model levels.

First we start with an example to present a usual scenario of concurrent development in MDE. In our examples we use diagrammatic specifications defined by means of DPF. The example is obviously simplified and only the details which are relevant for our discussion are presented. Then, common models, merge models and their computations are analysed in the subsequent sections.

Suppose that two software developers, Alice and Bob, use a version control system based on the copy-modify-merge paradigm. The scenario is depicted in Fig. 2, while an overview of the models in the example is shown in Fig. 3.

Alice checks out a local copy of the model V_1 (Fig. 1) from the repository and modifies it to V_{1A} , where 1 is a version number and A stands for Alice. In particular, she adds the node `PhDStudent` as an extension of `Student`, together with the arrow `enrols`. This modification takes place in the *evolution step* e_{1A} . Since the model in the repository may have been updated in the mean time, she needs to synchronise her model with the repository in order to integrate her local copy with other developers' modifications. This is done in the *synchronisation* s_{1A} . However, no modifications of the model V_1 has taken place in the repository while Alice was working on it. Therefore, the synchronisation completes without changing the local copy V_{1A} . Finally, Alice commits the local copy, which will be labelled V_2 in the repository (Fig. 3a). This is done in the *commit* c_{1A} .

Afterwards, Bob checks out a local copy of the model V_2 from the same repository and modifies it to V_{2B} . In particular, he takes into consideration also *Postdoc* as a different type of student; to avoid the pollution of extensions in the model he deletes the `PhDStudent` node, and refactors the model by adding a new node `Enrolment`. Then, he synchronises his model with the repository. Again, the synchronisation completes without changing the local copy V_{2B} . Finally, Bob commits the local copy, which will be labelled V_3 in the repository (Fig. 3b).

Alice continues working on her local copy, which is still V_2 and is not synchronised with the repository which contains Bob's modifications. She adds a node `Project`

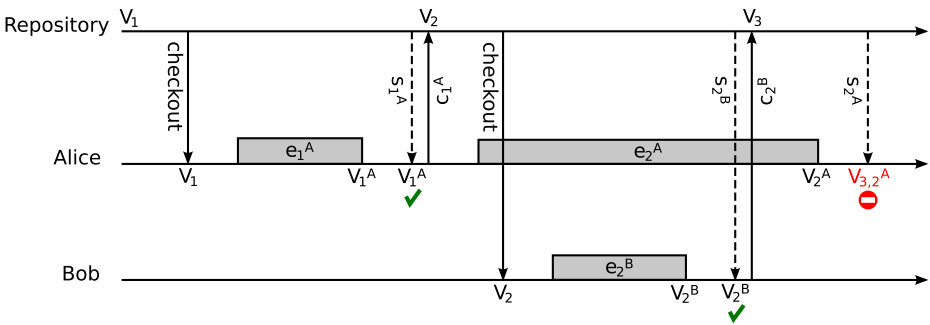


Fig. 2. The timeline of the example

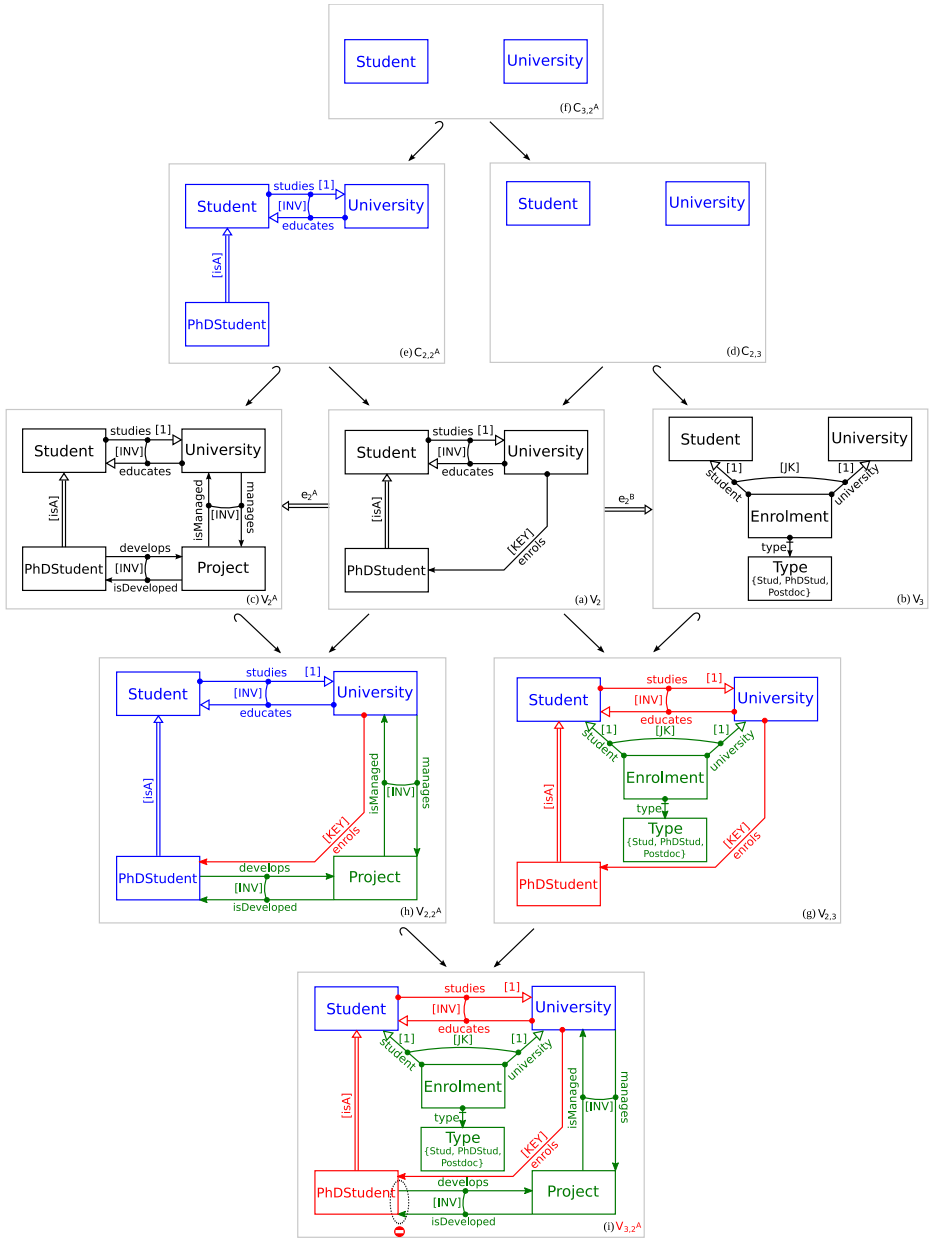


Fig. 3. The models of the example

(Fig. 3c). She synchronises her model with the repository where the last model is V_3 . Hence, the synchronisation computes the *merge model* $V_{3,2^A}$ (Fig. 3i). Now, the version control system reports a conflict in the merge model which forbids the commit C_{2^A} . This is because the node `PhDStudent` has been deleted by Bob, but Alice has added some arrows from/to it. The resolution of the conflict requires the manual intervention of Alice, who must review the model and decide to adapt it to Bob’s modifications, or, adapt Bob’s modifications to her own model.

3.1 Common Model

When Alice changes her local copy from V_2 to V_{2^A} , her development environment must keep track over what is common between the two models. The identification of what is common is the same as the identification of what is not modified, which should be feasible to implement in any tool.

Every two model elements which correspond to each other can be identified in a *common model*. For example, the model $C_{2,2^A}$ (Fig. 3e) is a common model of the models V_2 and V_{2^A} . The usage of a common model makes the construction of merge models at synchronisation step easy (explained in Sec. 3.2, 3.3). In some frameworks, what is common between two models is defined implicitly by stating that structurally equivalent elements imply that the elements are equal (*soft-linking*). This approach has the benefit of being general, but its current implementations are too resource greedy to be used in production environment. In other frameworks, elements with equal identifiers are seen as equal elements (*hard-linking*). Unfortunately, this approach is tool-dependent, since the element identification is different for every environment. Our claim is that “recording” which elements are kept unmodified during an evolution step addresses the problems of the soft- and hard-linking approaches. That is, these equalities are specified explicitly in common models as in the following definition (see Fig. 4).

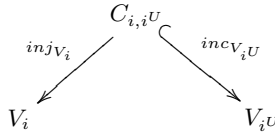


Fig. 4. Common model

Definition 4 (Common Model). A model C_{i,i^U} together with the injective morphism inj_{V_i} and the inclusion morphism $inc_{V_i^U}$ is a common model for V_i and V_i^U .

Note that we support renaming operations by allowing arbitrary injective morphisms inj_{V_i} . We decided, however, that the common model contains always the most recent names by requiring that the $inc_{V_i^U}$ are inclusions.

In order to find the common model between two models which are not subsequent versions of each other, i.e. for which we do not have a direct common model, we can construct the common model by the composition of the common models of their intermediate models. For example, the model $C_{3,2^A}$ (Fig. 3f) is the common model of the

models V_3 and V_{2A} . We call this common model for the *composition of commons* or the *normal form*. A possible way to compute this common model is as follows (see Fig. 5):

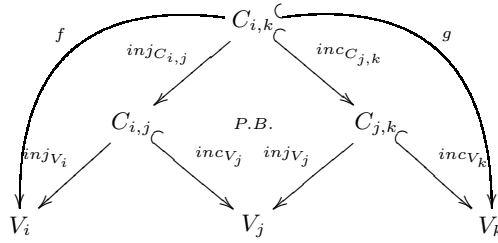


Fig. 5. Common models: $C_{i,j}$ and $C_{j,k}$; and the composition: $C_{i,k}$

Definition 5 (Composition of Commons). Given the diagrams $V_i \xleftarrow{inj_{V_i}} C_{i,j} \xrightarrow{inc_{V_j}} V_j$ and $V_j \xleftarrow{inj_{V_j}} C_{j,k} \xrightarrow{inc_{V_k}} V_k$ the common model for V_i and V_k is $C_{i,k}$ with the two morphisms f and g where $f = inj_{C_{i,j}}; inj_{V_i}$, $g = inc_{C_{j,k}}; inc_{V_k}$, and, $C_{i,k}$ is a pull-back ($C_{i,k}$, $inj_{C_{i,j}} : C_{i,k} \rightarrow C_{i,j}$, $inc_{C_{j,k}} : C_{i,k} \rightarrow C_{j,k}$) of the diagram $C_{i,j} \xrightarrow{inc_{V_j}} V_j \xleftarrow{inj_{V_j}} C_{j,k}$ such that $inc_{C_{j,k}}$ is an inclusion.

3.2 Merge Model

Recall that when Alice wanted to commit her local copy V_{2A} to the repository, she had to first synchronise it with the repository. In the synchronisation s_{2A} , a merge model $V_{3,2A}$ was created (Fig. 3i). The merge model must contain the information which is needed to distinguish which model elements come from which model. Since this is exactly one of the properties of pushout, we use pushout construction to compute merge models, as stated in the next definition (see Fig. 6).

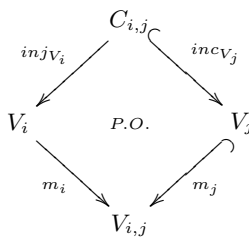


Fig. 6. Merge model

Definition 6 (Merge Model). Given the models V_i , V_j and $C_{i,j}$, the merge model $V_{i,j}$ is the pushout ($V_{i,j}$, $m_i : V_i \rightarrow V_{i,j}$, $m_j : V_j \rightarrow V_{i,j}$) of the diagram

$$V_i \xleftarrow{inj_{V_i}} C_{i,j} \xrightarrow{inc_{V_j}} V_j \text{ such that } m_j \text{ is an inclusion.}$$

The properties of the pushout are then used to decorate merge models such that added, deleted, moved, and renamed elements are distinguished (explained in Sec. 3.4).

3.3 Synchronisation and Commit

Fig. 7 outlines synchronisation and commit operations in the copy-modify-merge paradigm. These operations are defined as follows. In Fig. 7 and in the following definitions and propositions, U stands for “username”.

Definition 7 (Synchronisation). Given the local copy V_{iU} , the last model in the repository V_j and their merge model $V_{j,iU}$, the synchronisation $s_{iU} : (V_{iU}, V_j) \rightarrow V_{jU}$ is an operation which generates a synchronised local copy V_{jU} such that

$$V_{jU} := \begin{cases} V_{iU} & \text{if } i = j; \\ V_{j,iU} & \text{if } i < j, \text{ and } V_{j,iU} \notin \mathcal{C}^U \end{cases} \quad \text{with } \mathcal{C}^U \text{ the set of conflicting merge models.}$$

Definition 8 (Commit). Given the synchronisation $s_{iU} : (V_{iU}, V_j) \rightarrow V_{jU}$, the commit $c_{iU} : V_{jU} \Rightarrow V_{j+1}$ is an operation which adds the model V_{jU} to the repository as V_{j+1} .

Whenever a local copy V_{iU} is synchronised with a model V_j from the repository, if the version numbers are the same, i.e. $i = j$, then a *synchronised* local copy V_{jU} will be created such that $V_{jU} = V_{iU}$. However, if $i < j$, then a merge model $V_{j,iU}$ will be created such that $V_{jU} = V_{j,iU}$, only if $V_{j,iU}$ is not in a conflict state (explained in Sec. 3.4), i.e. $V_{j,iU} \notin \mathcal{C}^U$. Finally, the commit operation will add the synchronised local copy V_{jU} to the repository and will label it V_{j+1} . The next procedure explains the details of our approach to the synchronisation and commit operation (see Fig. 7).

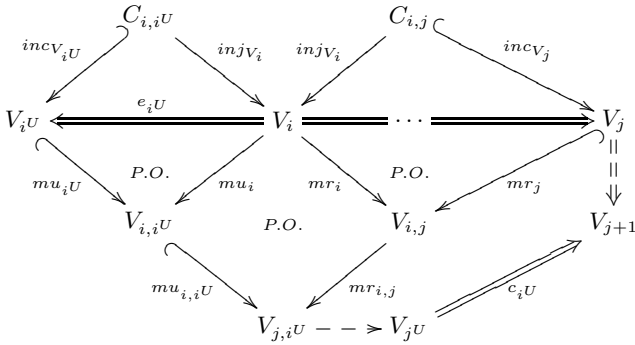


Fig. 7. Synchronisation and Commit

Procedure 9 (Synchronisation Procedure). Given the models V_{iU} , V_i , $C_{i,iU}$ and V_j , where $i < j$, the synchronisation $s_{iU} : (V_{iU}, V_j) \rightarrow V_{jU}$ is computed as follows:

1. compute the merge model $(V_{i,iU}, mu_{iU}, mu_i)$ as a pushout of

$$V_{iU} \xleftarrow{incV_{iU}} C_{i,iU} \xrightarrow{injV_i} V_i$$

2. compute the common model $(C_{i,j}, injV_i, incV_j)$ as a pullback of

$$V_i \xrightarrow{mr_i} V_{i,j} \xleftarrow{mr_j} V_j$$

3. compute the merge model $(V_{i,j}, mr_i, mr_j)$ as a pushout of

$$V_i \xleftarrow{inj_{V_i}} C_{i,j} \xrightarrow{inc_{V_j}} V_j$$

4. compute the merge model $(V_{j,i^U}, mu_{i,i^U}, mr_{i,j})$ as a pushout of

$$V_{i,i^U} \xleftarrow{mu_i} V_i \xrightarrow{mr_i} V_{i,j}$$

5. $V_{j^U} := V_{j,i^U}$ only if $V_{j,i^U} \notin \mathcal{C}^U$

3.4 Difference and Conflict

As mentioned, during a synchronisation operation $s_{i^U} : (V_{i^U}, V_j) \rightarrow V_{j^U}$ where $i < j$, the merge model V_{j,i^U} may contain conflicts. To detect these conflicts, we need a way to identify the differences between V_{i^U} and V_j , i.e. the modifications which has occurred in the evolution step(s). Difference identification in the merge model V_{j,i^U} can be done by distinguishing common elements, V_{i^U} -elements and V_j -elements from each other. However, since this is one of the properties of merge models, we already have all the information we need to identify the differences and, we only need a language to represent these differences. Moreover, since software models are graph-based structures, we need a diagrammatic language for this purpose. The language must enable tagging model elements as *common*, *added*, *deleted*, *renamed* and *moved*. We use DPF to define such a diagrammatic language, Δ , for the representation of model differences.


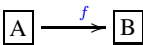

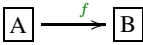

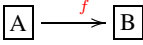
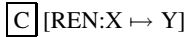
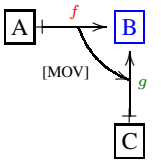
The language Δ is represented by the signature $\Sigma_\Delta = (\Pi_\Delta, \alpha_\Delta)$ which consists of five predicates: [common], [add], [delete], [rename], and [move] (see Table 3). The merge models will be *decorated* by predicates from the signature Σ_Δ in addition to the predicates from the signature which represents the modelling language.

Each of [common], [add] and [delete] has two arities: 1 and $1 \xrightarrow{x} 2$. That is, each of these predicates can be used to tag either a node or an arrow. For example, Bob has added the node `Enrolment` and the arrows `student` and `university` in the model V_3 (Fig. 3b). These added elements are coloured green, i.e. tagged as *added*, in the merge model $V_{2,3}$ (Fig. 3g) since the visualisation of the predicate [add] in Σ_Δ is green.

For the predicate [rename], when an element $A \in V_i$ is renamed to $B \in V_{i^U}$, the common model C_{i,i^U} will contain B with $inj_{V_i}(B) = A$ and $inc_{V_{i^U}}(B) = B$. Moreover, the visualisation will be \boxed{B} [REN:A \mapsto B] in the merge model V_{i,i^U} . The morphism inj_{V_i} is injective in order to allow for this renaming. Moreover, the morphism $inc_{V_{i^U}}$ is inclusion so that the common- and the merge models always contains the new name. However, when V_{j,i^U} is a merge model for $j > i$, then the visualisation will be \boxed{C} [REN:C \mapsto Y], where $C \in V_i$ is the old name, and $Y \in V_j$ or $Y \in V_{i^U}$ is the new name. This is due to the commutative property of pushouts. Fig. 8 shows an example of renaming, where `Employee` is renamed in an evolution step e_{1^A} to `Person`.

In general, the predicate [move] is used when the source of the reference to a contained model element is changed from a container to another. In object oriented models, e.g. in class diagrams, this operation is usually used in two cases; when an attribute or a method of a class is moved to another class, and, when a class is moved from a package to another. An example of the usage of a move operation is shown in Fig. 8, where the attribute `salary` is moved to the *objectified* relationship `Employment`.

Table 3. The signature Σ_Δ . $A, B, C, f, g \in V_{i,j}$. $V_{i,j}$ and $C_{i,j}$ are the merge and common models, respectively, of V_i and V_j , with $i < j$.

Π_Δ	α_Δ	Proposed visualisation	Intended semantics
$[\text{common}]^n$	1		$A \in V_i$ and $A \in V_j$
$[\text{common}]^a$	$1 \xrightarrow{x} 2$		$f \in V_i$ and $f \in V_j$
$[\text{add}]^n$	1		$A \notin V_i$ and $A \in V_j$
$[\text{add}]^a$	$1 \xrightarrow{x} 2$		$f \notin V_i$ and $f \in V_j$
$[\text{delete}]^n$	1		$A \in V_i$ and $A \notin V_j$
$[\text{delete}]^a$	$1 \xrightarrow{x} 2$		$f \in V_i$ and $f \notin V_j$
$[\text{rename}]$	1		$\exists C \in C_{i,j} : \text{inj}_{V_i}(C) = X$ and $\text{inc}_{V_j}(C) = Y$ where $\text{inj}_{V_i} : C_{i,j} \rightarrow V_i$ and $\text{inc}_{V_j} : C_{i,j} \rightarrow V_j$
$[\text{move}]$	$1 \xrightarrow{x} 2$ $\uparrow y$ 3		$f \in V_i$ and $f \notin V_j$ and $g \in V_j$ and $g \notin V_i$ and $B \in C_{i,j}$ and both f and g are containment arrows as defined in Table 1

The synchronisation procedure we have developed uses Σ_Δ for two main purposes:

- to *reduce* the decorated merge model V_{j,i^U} according to the rules in Table 4, e.g. if a model element is tagged with both $[\text{common}]$ and $[\text{delete}]$, it will be tagged only with $[\text{delete}]$ in V_{j,i^U} .
- to obtain the synchronised local copy V_{j^U} from V_{j,i^U} by *interpreting* the predicates as operations, e.g. if a model element is tagged with the predicate $[\text{delete}]$, it will not exist in V_{j^U} (see Table 4).

If the reduced merge model V_{j,i^U} contains the predicate $[\text{conflict}]$, then $V_{j,i^U} \in \mathcal{C}^U$, i.e. it is in a state of conflict. Although conflicts are context-dependent, we have recognised some situations where syntactic conflicts will arise. The definition of new rules/conflicting situations is also allowed in DPF. The following is a summary of the concurrent modifications which we identify as conflicts:

- adding structure to an element which has been deleted
- renaming an element which has been renamed
- moving an element which has been moved

In Table 4, the predicates in $V_{j,i^U}(\Pi_\Delta)$ are written in the form $([p], \delta_x^v)$ with v a version number and $p \in \Pi_\Delta$, where v is used to distinguish between predicates which come from V_{i,i^U} and $V_{i,j}$ (see Def. 2). For example, $[\text{common}](X)$ in the first column is an abbreviation for $(([\text{common}]^n, \delta_x^{i^U}) : 1 \mapsto X) \in V_{j,i^U}(\Pi_\Delta)$ for $x \in \mathbb{N}$. That

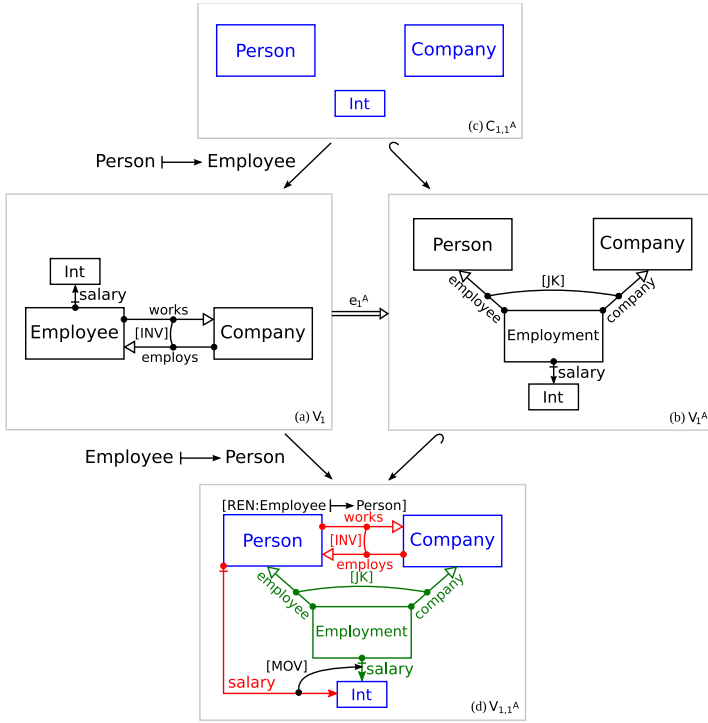


Fig. 8. Examples of the predicates [move] and [rename]

Table 4. A subset of the rules used for the reduction of $V_{j,i}U$ and to obtain V_{jU} from $V_{j,i}U$. $X, f \in V_{j,i}U$

$V_{i,i}U(\Pi_{\Delta})$	$V_{i,j}(\Pi_{\Delta})$	$V_{j,i}U(\Pi_{\Delta})$	In V_{jU}
[common] (X)	[common] (X)	[common] (X)	remains
[delete] (X)	[delete] (X)	[delete] (X)	deleted
[common] (X)	[delete] (X)	[delete] (X)	deleted
[delete] (X)	[common] (X)	[delete] (X)	deleted
[move] (X)	[move] (X)	[conflict] (X)	\perp
[add] ^a (f)	[delete] ⁿ ($src(f)$)	[conflict] ($f, src(f)$)	\perp
[add] ^a (f)	[delete] ⁿ ($trg(f)$)	[conflict] ($f, trg(f)$)	\perp
[rename] (X)	[rename] (X)	[conflict] (X)	\perp

is, the predicate $[\text{common}]^n$ comes from the model $V_{i,iU}$. Moreover, \perp means that the synchronised local model V_{jU} will not be created.

4 Related Work

The literature related to model evolution and in particular version control is becoming abundant. Firstly, we have the works that describe how to compute the difference of models: EMF Compare [2] and DSMDiff [11] are two model differencing tools which are based on a similar technique. The difference calculation is divided in two phases. The first is the detection of model mappings, where all the elements of the two input models are compared using metrics like signature matching and structural similarity. The second phase is the determination of model differences, where all the additions, deletions and changes are detected. This approach has the great benefit of being general, but at the price of being resource greedy.

Secondly, there are works which analyse how to represent differences among models conforming to an arbitrary metamodel. There are different approaches for the representation of model differences:

1. As models which conform to a difference metamodel. The difference metamodel can be generic [12], or obtained by an automated transformation [3]. Those models are in general minimalistic (i.e. only the necessary information to represent the difference is presented), transformative (i.e. each difference model induce a transformation), compositional (i.e. difference models can be composed sequentially or in parallel), and typically symmetric (i.e. given a difference representation we can compute the inverse of it).
2. As a model which is the union of the two compared models, with the modified elements highlighted by colours, tags, or symbols [13], which is similar to our visualisation. The adoption of this technique is typically beneficial for the designer, since the rationale of the modifications is easily readable. However, these quality factors are retained only if the base models are not large and not too many updates apply to the same elements, since the difference model consists of both base models to denote the differences.
3. As a sequence of atomic actions specifying how the initial model is procedurally modified [14]. While this technique has the great advantage of being very efficient, the difference representation is not readable and intuitive. In addition, edit scripts do not follow the “everything is a model vision”. They are suitable for internal representations but quite ineffective to be adopted for documenting changes in MDE environments.

Thirdly, there are works aimed at identifying the types of structural and semantic conflicts that can occur in distributed development. In [4] a predefined set of *a priori* conflicts is identified, stating that it is not possible to provide a generic technique for conflict detection with an arbitrary accuracy. However, in [15,16] the authors propose a Domain-Specific Modelling Language for the definition of weaving models which represent custom conflicting patterns. Moreover, it is possible to describe the resolution criteria through OCL expressions.

5 Conclusion and Future Work

In this paper, category-theoretical constructs are used to formalise concepts used in version control. Usual operations such as checkout, synchronise and commit that a developer perform in a distributed development are analysed. Moreover, the concepts of common and merge models are introduced and defined as pullback and pushout, respectively. In addition we defined a language Δ – specified as the signature Σ_{Δ} in DPF – which we have used to formalise model differences. The predicates of Σ_{Δ} enable the reasoning about and presentation of operations such as add, delete, move, and rename. That is, model elements which has been added, deleted, moved or renamed are tagged by predicates from Σ_{Δ} . Finally, we described how these predicates can be used for the identification of possible conflicting modifications. DPF has shown to have the expressiveness and flexibility which are required to define the language Δ .

The proposed approach to version control in MDE differs from the approaches in the related work mentioned above since it is based on common models instead of difference models. The difference between two models is identified by means of category-theoretical constructs and represented through the language Δ .

In this work, we focused only on the detection of a predefined set of syntactic conflicts which are derived from experience. In a future work, we analyse and formalise semantic conflicts, i.e. modifications which are violating metamodel constraints or predicate dependencies. Moreover, a prototype implementation of these techniques will be necessary to show the efficiency of the proposed techniques. This is a challenging task, considering the lack of mature standards and the issues related to the identification of model elements [17].

References

1. Baresi, L., Heckel, R.: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 431–433. Springer, Heidelberg (2004)
2. Brun, C., Musset, J., Toulmé, A.: EMF Compare Project, <http://www.eclipse.org/emft/projects/compare/>
3. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology* 6(9), 165–185 (2007) (Special Issue on TOOLS Europe 2007)
4. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science* 127(3), 113–128 (2005)
5. Niu, N., Easterbrook, S., Sabetzadeh, M.: A Category-theoretic Approach to Syntactic Software Merging. In: *ICSM 2005: 21st IEEE International Conference on Software Maintenance*, pp. 197–206. IEEE Computer Society, Los Alamitos (2005)
6. Rutle, A., Wolter, U., Lamo, Y.: Diagrammatic Software Specifications. In: *NWPT 2006: 18th Nordic Workshop on Programming Theory* (October 2006)
7. Rutle, A., Wolter, U., Lamo, Y.: A Diagrammatic Approach to Model Transformations. In: *EATIS 2008: Euro American Conference on Telematics and Information Systems* (2008)
8. Rutle, A., Wolter, U., Lamo, Y.: A Formal Approach to Modeling and Model Transformations in Software Engineering. Technical Report 48, Turku Centre for Computer Science, Finland (2008)

9. Wolter, U., Diskin, Z.: Generalized Sketches: Towards a Universal Logic for Diagrammatic Modeling in Software Engineering. In: ACCAT Workshop 2007, satellite event of ETAPS 2007: European Joint Conferences on Theory and Practice of Software (to appear)
10. Diskin, Z.: Model Transformation via Pull-backs: Algebra vs. Heuristics. Technical Report 521, School of Computing, Queen's University, Kingston, Canada (September 2006)
11. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems* 16(4), 349–361 (2007) (Special Issue on Model-Driven Systems Development)
12. Rivera, J.E., Vallecillo, A.: Representing and Operating with Model Differences. In: 46th International Conference on TOOLS Europe 2008: Objects, Components, Models and Patterns. LNBP, vol. 11, pp. 141–160. Springer, Heidelberg (2008)
13. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. In: ESEC/FSE 2003: 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003, pp. 227–236. ACM, New York (2003)
14. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
15. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
16. Cicchetti, A., Rossini, A.: Weaving Models in Conflict Detection Specifications. In: SAC 2007: 22nd ACM Symposium on Applied Computing, pp. 1035–1036. ACM, New York (2007)
17. Rutle, A., Rossini, A.: A Tentative Analysis of the Factors Affecting the Industrial Adoption of MDE. In: ChaMDE 2008: 1st International Workshop on Challenges in Model-Driven Software Engineering, pp. 57–61 (2008), http://sse1.vub.ac.be/ChaMDE08/_media/chamde2008_proceedings.pdf