A Formalisation of the Copy-Modify-Merge Approach to Version Control in MDE

Alessandro Rossini\textsuperscript{a}, Adrian Rutle\textsuperscript{b}, Yngve Lamo\textsuperscript{b}, Uwe Wolter\textsuperscript{a}

\textsuperscript{a}University of Bergen, P.O. Box 7803, 5020 Bergen, Norway
\textsuperscript{b}Bergen University College, P.O. Box 7030, 5020 Bergen, Norway

Abstract

Models are the primary artefacts of the software development process in Model-Driven Engineering (MDE). Like other software artefacts, models undergo a complex evolution during their life cycles. Version control is one of the key techniques which enable developers to tackle this complexity. Traditional version control systems (VCS) are based on the copy-modify-merge approach which is not fully exploited in MDE since current implementations lack model-orientation. In this paper we provide a formalisation of the copy-modify-merge approach in the context of MDE. In particular, we analyse how the identification of commonalities and the calculation of differences can be defined by means of category-theoretical constructions. Moreover, we demonstrate how the properties of these constructions can be used to synchronise models and detect conflicting modifications.

Keywords: Model-Driven Engineering, Version Control System, Category Theory, Diagram Predicate Framework

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 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, there is a growing need for techniques and tools to support model evolution activities such as version control.

Present-day MDE tools offer limited support for the version control of models. Typically, this problem is addressed using a lock-modify-unlock approach [1], where a repository allows only one developer to work on a particular 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 spread, dealing with locking issues may become problematic.

In contrast, traditional version control systems (VCS) such as Subversion facilitate efficient concurrent development of source code. These systems are based on the copy-modify-merge approach [1]. In this approach, each developer accesses a repository and creates a personal working copy – a snapshot of the repository’s files and directories. Then, the developers modify their working copies simultaneously and independently. Finally, the local modifications are merged together into the repository. The VCS assists in the merging by detecting conflicting modifications. When a conflict is detected, the system requires manual intervention from the developer.

Unfortunately, traditional VCSs focus on the management of text-based files, such as source code. Thus, the difference calculation, merging of modifications and conflict detection are based on a per-line
textual comparison. Since the structure of models is graph-based rather than text- or tree-based, the existing techniques are not suitable for MDE.

Recent research has led to a number of findings in model evolution. The interested reader may consult [2] for difference calculation, [3] for difference representation, [4] for conflict detection and [5] for a survey on software merging, to cite a few. However, the proposed solutions are not formalised enough to enable automatic reasoning about model evolution. For example, operations such as change or update are given different and ambiguous semantics in different works. Moreover, the terminology used in these solutions is not precise, e.g. terms “add”, “create” and “insert” are often used to refer to the same operations. Furthermore, the approach to version control (e.g. copy-modify-merge) is not explicitly stated and concepts such as synchronisation and commit are only defined semi-formally.

Our claim is that the copy-modify-merge approach must be adopted to enable effective version control in MDE. This would require formal techniques which target graph-based structures. The goal of this paper is the formalisation of the copy-modify-merge approach in MDE. In particular, we demonstrate that the identification of commonalities and the calculation of differences can be defined as pullback and pushout constructions, respectively. For our analysis we use the Diagram Predicate Framework (DPF) [6] which provides a formal diagrammatic approach to modelling based on category theory [7]. DPF is a generalisation and adaptation of the categorical sketch formalism, where user-defined diagrammatic predicate signatures are allowed to represent the constructs of modelling languages in a more direct and adequate way. In particular, DPF is an extension of the Generalised Sketches [8] formalism originally developed by Diskin et al. in [9, 10, 11]. A detailed discussion of the advantages of Generalized Sketches in software engineering and the reasons for choosing it rather than traditional Sketches can be found in [12].

This paper further develops the work on the formalisation of version control in MDE already published in [13]. Firstly, we have reorganised the structure of the paper and added several examples from software engineering which help the reader to gain insight into our reasoning. Secondly, we have extended the theoretical foundation by defining the concepts on which our approach to version control relies. Thirdly, we have described the synchronisation procedure in more detail and enriched the terminology related to these details. Finally, we have formalised the rules used for conflict detection.

The content of this paper is neither purely theoretical nor purely practical; rather it seeks to bridge the gap between these worlds. We provide a formal approach to version control motivated and illustrated by practical examples. We introduce only the theoretical elements which are necessary to investigate, formalise, and solve the practical problems. More precisely, we explicitly define the formal concepts and constructions needed in order to understand the paper, such as graph, graph homomorphism, category, pullback and pushout. For a more comprehensive discussion of these concepts and constructs, the interested reader is encouraged to consult the literature, for example [7, 14].

The remainder of the paper is structured as follows. Section 2 provides an introduction to DPF and to the graph-theoretical and category-theoretical concepts used in the paper. Section 3 outlines an illustrative example and provides the formalisation of the concepts of version control. In Section 4 the state-of-the-art research in version control is summarised. Finally, in Section 5 some concluding remarks and ideas for future work are offered.
2. Diagram Predicate Framework

The present paper examines the contribution of DPF in version control in the context of MDE. A detailed discussion of DPF’s usage in the formalisation of concepts in (meta)modelling and model transformations is given in [6] and [15, 16], respectively.

DPF is a diagrammatic specification formalism that takes its main ideas from graph theory, category theory and first-order logic (FOL), and adapts them to software engineering needs. While in FOL the arity of a predicate is given by a collection of nodes, it is given by a graph in DPF; i.e. a collection of nodes linked by a collection of arrows. The main difference between FOL and DPF is that FOL is an “element-wise” logic; i.e. variables vary over elements of sets. In contrast, DPF uses a “sort-wise” logic, in which variables vary over sets and mappings, respectively.

We understand the term “diagrammatic specification formalism” as a type of formalism that is targeting graph-based structures. Although graph-based structures are often visualised in a natural way, we do not treat “diagrammatic” and “visual” as synonyms. In particular, we have to bear in mind that it may be a difficult task, and sometimes even impossible, to find appropriate and intuitive visualisations for all aspects of a diagrammatic specification formalism.

In DPF, software models are represented by *diagrammatic specifications*. In the remainder of the paper, we use the terms “model” and “diagrammatic specification” interchangeably. Diagrammatic specifications can be based on any kind of graph structures (see [12] for the general case). However, the version of DPF which we develop in this paper is based on directed multi-graphs. In the following, we introduce some basic graph-theoretic concepts. The notation is adopted from [7].

**Definition 1 (Graph).** A graph $G = (G_0, G_1, \text{src}^G, \text{trg}^G)$ is given by a collection $G_0$ of nodes, a collection $G_1$ of arrows and two maps $\text{src}^G : G_1 \rightarrow G_0$ assigning the source and target to each arrow, respectively. We write $f : X \rightarrow Y$ to indicate that $\text{src}(f) = X$ and $\text{trg}(f) = Y$.

**Definition 2 (Subgraph).** A graph $G = (G_0, G_1, \text{src}^G, \text{trg}^G)$ is a subgraph of a graph $H = (H_0, H_1, \text{src}^H, \text{trg}^H)$, written $G \subseteq H$, iff $G_0 \subseteq H_0$, $G_1 \subseteq H_1$ and $\text{src}^G(f) = \text{src}^H(f)$, $\text{trg}^G(f) = \text{trg}^H(f)$, for all $f \in G_1$.

**Definition 3 (Graph Homomorphism).** A graph homomorphism $\varphi : G \rightarrow H$ is a pair of maps $\varphi_0 : G_0 \rightarrow H_0$, $\varphi_1 : G_1 \rightarrow H_1$ which preserve the sources and targets; i.e. for each arrow $f : X \rightarrow Y$ in $G$ we have $\varphi_1(f) : \varphi_0(X) \rightarrow \varphi_0(Y)$ in $H$.

**Remark 1 (Inclusion Graph Homomorphism).** $G \subseteq H$ iff the inclusion maps $\text{inc}_0 : G_0 \hookrightarrow H_0$, $\text{inc}_1 : G_1 \hookrightarrow H_1$ define a graph homomorphism $\text{inc} : G \hookrightarrow H$.

**Definition 4 (Category of Graphs).** The category $\text{Graph}$ has graphs as objects and its morphisms are graph homomorphisms. The composition $\psi \cdot \varphi : G \rightarrow K$ of two graph homomorphisms $\varphi : G \rightarrow H$ and $\psi : H \rightarrow K$ is defined component-wise $\psi \cdot \varphi = (\varphi_0 \cdot \psi_0, \varphi_1 \cdot \psi_1)$. The identity graph homomorphisms $\text{id}_G : G \rightarrow G$ are also defined component-wise $\text{id}_G = (\text{id}_{G_0}, \text{id}_{G_1})$. This ensures that the composition of graph homomorphisms is associative and that identity graph homomorphisms are identities with respect to composition. By $\text{Graph}_0$ we denote the collection of all objects in this category; i.e. the collection of all graphs.

2.1. Syntax of Diagrammatic Specification

A diagrammatic specification is a structure which consists of a graph decorated by a set of constraints. The graph represents the structure of the model. Predicates from a predefined diagrammatic signature are used to add constraints to the graph. The formal definitions are as follows:

**Definition 5 (Signature).** A (diagrammatic predicate) signature $\Sigma := (\Pi, \alpha)$ consists of a collection of predicate symbols $\Pi$ with a mapping $\alpha : \Pi \rightarrow \text{Graph}_0$ that assigns a graph to each predicate symbol $p \in \Pi$. Accordingly, $\alpha(p)$ is called the arity of the predicate symbol $p$. 

3
**Definition 6 (Constraint).** Given a signature $\Sigma = (\Pi, \alpha)$, a constraint $(p, \delta)$ on a graph $G$ is given by a predicate symbol $p$ and a graph homomorphism $\delta : \alpha(p) \to G$.

**Definition 7 (Diagrammatic Specification).** Given a signature $\Sigma = (\Pi, \alpha)$, a $\Sigma$-specification $S := (G^S, \Gamma^S)$ is given by a graph $G^S$ and a set $\Gamma^S$ of constraints $(p, \delta)$ on $G^S$ with $p \in \Pi$.

**Definition 8 (Subspecification).** A $\Sigma$-specification $S := (G^S, \Gamma^S)$ is a $\Sigma$-subspecification of a $\Sigma$-specification $T := (G^T, \Gamma^T)$, written $S \subseteq T$, iff $G^S \subseteq G^T$ and $\Gamma^S \subseteq \Gamma^T$.

Although the intention of this paper is not to deal with the semantic aspects of models and version control, we have included some remarks concerning semantics in this section. In order to formalise a particular modelling environment, we have to define a corresponding diagrammatic predicate signature. In addition, we have to make an appropriate choice regarding the semantics of nodes, arrows and predicates. In formalisations of object-oriented modelling, for example, it is appropriate to interpret nodes as sets and arrows as multi-valued functions.

Fig. 1 shows an example of a UML class diagram [17] which specifies the structural model of a simple information system for universities. This model includes two classes and an association between them. The association does not specify the relation between the two classes; it defines two multi-valued functions $uS\text{tuds}$ and $s\text{Univ}$, both of which represent the same relation. These functions are also called references in object-oriented modelling and they reflect the way in which associations are actually implemented. Moreover, the multiplicity constraints refer directly to these two functions.

![UML class diagram](image)

Figure 1: A UML class diagram $S^{UML}$

Throughout the paper, we will interpret nodes as sets and arrows $X \xrightarrow{f} Y$ as multi-valued functions $f : X \to \wp(Y)$. The powerset $\wp(Y)$ of $Y$ is the set of all subsets of $Y$; i.e. $\wp(Y) = \{X | X \subseteq Y\}$. Moreover, the composition of two multi-valued functions $f : X \to \wp(Y)$, $g : Y \to \wp(Z)$ is defined by the union of image sets $(f; g)(x) := \bigcup\{g(y) | y \in f(x)\}$.

Table 1 shows a sample signature $\Sigma = (\Pi, \alpha)$. The first column of the table shows the names of the predicates. The second and the third columns show the arities of predicates and a possible visualisation of the corresponding constraints, respectively. In the fourth column, the intended semantics of each constraint is specified. The predicates in Table 1 allow us to specify constraints that, according to our analysis, are useful for structural modelling. For example the $\text{injective}$ predicate is used to denote an injective function. Moreover, the $\text{jointly injective}$ predicate is used to denote the fact that two functions form a tabulation [18]. The sample signature $\Sigma$, including the proposed visualisation, has been applied later in the paper to present our examples.
Table 1: A sample signature $\Sigma$

<table>
<thead>
<tr>
<th>Property</th>
<th>$\alpha$</th>
<th>Proposed visualisation</th>
<th>Intended semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\text{mult}(n,m)]$</td>
<td>$1 \xrightarrow{f} 2$</td>
<td>$\forall x \in X : n \leq</td>
<td>f(x)</td>
</tr>
<tr>
<td>$[\text{inverse}]$</td>
<td>$1 \xrightarrow{f} 2$</td>
<td>$\forall x \in X, \forall y \in Y : y \in f(x) \iff x \in g(y)$</td>
<td></td>
</tr>
<tr>
<td>$[\text{inj}]$</td>
<td>$1 \xrightarrow{f} 2$</td>
<td>$\forall x, x' \in X : f(x) = f(x') \implies x = x'$</td>
<td></td>
</tr>
<tr>
<td>$[\text{surj}]$</td>
<td>$1 \xrightarrow{f} 2$</td>
<td>$f(X) = Y$</td>
<td></td>
</tr>
<tr>
<td>$[\text{surj}] \times [\text{inj}]$</td>
<td>$1 \xrightarrow{f} 2$</td>
<td>$f(X) \cup f(Z) = Y$</td>
<td></td>
</tr>
</tbody>
</table>

Example 1. Fig. 2a shows an example of a $\Sigma$-specification $S = (G^S, \Gamma^S)$. $S$ represents the same information as the UML class diagram in Fig. 1. Fig. 2b presents the underlying graph $G^S$ of $S$; i.e. the graph of $S$ without any constraints. In $S$, every university educates one or more students; this is forced by the constraint ($[\text{mult}(1,\infty)], \delta_1$) on the arrow $u \xrightarrow{[\text{stud}]}$ (see Table 2). Moreover, every student studies at exactly one university; this is forced by the constraint ($[\text{mult}(1,1)], \delta_2$) on the arrow $s \xrightarrow{[\text{univ}]}$. Another property of $S$ is that the functions $sUniv$ and $uStuds$ are inverse of each other; i.e. $\forall u \in University$ and $\forall s \in Students : u \in sUniv(s) \iff s \in uStuds(u)$. This is forced by the constraint ($[\text{inverse}], \delta_1$) on $sUniv$ and $uStuds$.

Figure 2: A diagrammatic specification $S = (G^S, \Gamma^S)$ and its underlying graph $G^S$. 


2.2. Category of Specifications

In order to describe the relations between models in VCSs, we have introduced the concept of specification morphisms.

**Definition 9 (Specification Morphism).** A \( \Sigma \)-specification morphism \( f : S \rightarrow T \) between two \( \Sigma \)-specifications \( S = (G^S, \Gamma^S) \) and \( T = (G^T, \Gamma^T) \) is a graph homomorphism \( f : G^S \rightarrow G^T \) preserving constraints; i.e. \( (p, \delta) \in \Gamma^S \) implies \( (p, \delta; f) \in \Gamma^T \) for all constraints \( (p, \delta) \in \Gamma^S \).

**Remark 2 (Inclusion Specification Morphism).** \( S \subseteq T \) if and only if the inclusion graph homomorphism \( \text{inc} : G^S \hookrightarrow G^T \) defines a \( \Sigma \)-specification morphism \( \text{inc} : S \rightarrow T \).

For any signature \( \Sigma \), we obtain a category \( \text{Spec}^\Sigma \) of all \( \Sigma \)-specifications and all \( \Sigma \)-specification morphisms. The associativity of graph homomorphism composition ensures that (i) the composition of two \( \Sigma \)-specification morphisms becomes a \( \Sigma \)-specification morphology as well, and (ii) the composition of \( \Sigma \)-specification morphisms is also associative. Furthermore, the identity graph homomorphisms \( \text{id}^G : G^S \rightarrow G^S \) and \( \text{id}^T : G^T \rightarrow G^T \) define identity \( \Sigma \)-specification morphisms \( \text{id}^S : S \rightarrow S \), which are neutral with respect to composition.

Bearing in mind that the construction of limits and colimits in the category of graphs is based on the corresponding component-wise constructions in the category of sets [14], it is possible to extend limit and colimit construction for graphs to the corresponding construction for specifications.

In this paper, we consider addition, deletion and renaming to be modifications in model evolution (see Section 3.2). It is enough, therefore, to consider only injective and inclusion specification morphisms. To make the paper self-contained, we present the corresponding versions of pullback and pushout constructions which are used in our approach to version control.

**Proposition 1 (Pullback).** Given \( \Sigma \)-specifications \( S, T, M \) and two injective \( \Sigma \)-specification morphisms \( m : T \rightarrow M \) and \( n : S \rightarrow M \), we can construct a \( \Sigma \)-specification \( C \) and an injective \( \Sigma \)-specification morphism \( m^* : C \rightarrow S \) such that \( C \subseteq T \) and the resulting diagram is commutative and is also a pullback in \( \text{Spec}^\Sigma \).
Hence, the graph $G_C$ is defined as follows:

$$G_C^i := \{ x \in G^T_i \mid m_i(x) \in n_i(G_S^i) \}, \ i = 0, 1$$

$$\text{src}^{G_C}(f) := \text{src}^{G_T}(f) \quad \text{for all } f \in G_C^i$$

$$\text{trg}^{G_C}(f) := \text{trg}^{G_T}(f) \quad \text{for all } f \in G_C^i$$

The graph homomorphism $m^* : G_C \to G_S$ is given by the following, for $i = 0, 1$

$$m^*_i(x) := m_i(x) \quad \text{for all } x \in G_C^i$$

Moreover, we have

$$\Gamma_C := \{(p, \delta) \in \Gamma^T \mid \exists (p, \sigma) \in \Gamma^S \text{ with } \delta; m = \sigma; n\}$$

**Remark 3 (Uniqueness of Pullback).** The pullback $(C, m^* : C \to S, n^* : C \leftarrow T)$ in Proposition 1 is unique due to the inclusion $n^*$.

**Proposition 2 (Pushout).** Given $\Sigma$-specifications $C, S, T$ and injective $\Sigma$-specification morphisms $n : C \to T, m : C \to S$, we can construct a $\Sigma$-specification $M$ and injective $\Sigma$-specification morphisms $n^* : S \to M, m^* : T \to M$, such that the resulting diagram is commutative and a pushout in $\text{Spec}^\Sigma$.

Hence, the graph $G_M$ is defined as follows:

$$G_M^i := \{ (S.x \mid x \in G^T_i, x \notin m_i(G_S^i)) \cup G_C^i \cup \{ T.x \mid x \in G^T_i, x \notin n_i(G_C^i) \}, \ i = 0, 1$$

$$\text{src}^{G_M}(f) := \begin{cases} y, & \text{if } f = S.g, x = \text{src}^{G_T}(g) \in m_0(G_C^0), m_0(y) = x \\ S.x, & \text{if } f = S.g, x = \text{src}^{G_T}(g) \notin m_0(G_C^0) \\ \text{src}^{G_C}(f), & \text{if } f \in G_C^1 \\ y, & \text{if } f = T.g, x = \text{src}^{G_T}(g) \in n_0(G_C^0), n_0(y) = x \\ T.x, & \text{if } f = T.g, x = \text{src}^{G_T}(g) \notin n_0(G_C^0) \end{cases}$$

$$\text{trg}^{G_M}(f) \text{ is defined analogously}$$
The graph homomorphisms $n^*: G^S \to G^M$, $m^*: G^T \to G^M$ are given by the following, for $i = 0, 1$

$$n_i^*(x) := \begin{cases} y, & \text{if } x \in m_i(G^C_i), \ m_i(y) = x \\ S.x, & \text{if } x \notin m_i(G^C_i) \end{cases}$$

$$m_i^*(x) := \begin{cases} y, & \text{if } x \in n_i(G^C_i), \ n_i(y) = x \\ T.x, & \text{if } x \notin n_i(G^C_i) \end{cases}$$

Moreover, we have

$$\Gamma^M := \{(p, \delta; n^*) \mid (p, \delta) \in \Gamma^S\} \cup \{(q, \sigma; m^*) \mid (q, \sigma) \in \Gamma^T\}$$

**Remark 4 (Uniqueness of Representatives).** The $y$'s in Proposition 2 are uniquely determined since the morphisms $m_i$ and $n_i$, with $i = 0, 1$ are assumed to be injective.

**Remark 5 (Pushout for Inclusions).** The notation $S.x$ refers to the element $x$ from specification $S$, where $S$ is considered the name (unique identifier) of the specification. This notation is used to resolve possible name conflicts; i.e. to ensure disjoint union. However, if $C \sqsubseteq T$ there will not be name conflicts between $C$ and $T$. In this case we can simplify the construction of $M$, thus the morphism $m^*$ becomes an inclusion.

The graph $G^M$ is defined as follows:

$$G_i^M := \{S.x \mid x \in G^S_i, \ x \notin m_i(G^C_i)\} \cup G^T_i, \ i = 0, 1$$

$$\text{src}^{G^M}(f) := \begin{cases} y, & \text{if } f = S.g, x = \text{src}^{G^S}(g) \in m_0(G^C_0), \ m_0(y) = x \\ S.x, & \text{if } f = S.g, x = \text{src}^{G^S}(g) \notin m_0(G^C_0) \\ \text{src}^{G^T}(f), & \text{if } f \in G^T_1 \end{cases}$$

$$\text{trg}^{G^M}(f)$$ is defined in the same way

Moreover, we have

$$\Gamma^M := \{(p, \delta; n^*) \mid (p, \delta) \in \Gamma^S\} \cup \Gamma^T$$

**Remark 6 (Identification of Constraints).** Two constraints $(p, \delta) \in \Gamma^S$ and $(p, \sigma) \in \Gamma^T$ such that $\delta;m^* = \sigma; n^*$ are mapped to the same constraint $(p, \delta; m^*) = (p, \sigma; n^*) \in \Gamma^M$. More precisely, we obtain for all constraints $(r, \gamma) \in \Gamma^C$ just one constraint $(r, \gamma; n; m^*) = (r, \gamma; m; n^*) \in \Gamma^M$. 

8
2.3. Hierarchical Relationships

Any object-oriented modelling language must contain special constructs to express hierarchical relationships of the kind “extension” [19]. In object-oriented modelling, when $X'$ is said to extend $X$, it means that any instance of $X'$ is considered also an instance of $X$. More precisely, $X'$ inherits all attributes and references from $X$. We can express this kind of inheritance within DPF by requiring that specifications have certain "global" properties.

**Definition 10 (Extension).** Given a $\Sigma$-specification $S = (G^S, \Gamma^S)$, an arrow $e : X' \rightarrow X$ in $G^S$ is an extension if the following properties are satisfied:

1. To any arrow $f : X \rightarrow Y$ in $G^S$ we can assign an arrow $f' : X' \rightarrow Y$ in $G^S$ with $f' = e; f$.
2. This assignment is injective; i.e. for all $f_1, f_2 : X \rightarrow Y$ with $f_1 \neq f_2$ implies $f'_1 \neq f'_2$.
3. The arrow $e$ is declared to be total, single-valued and injective; i.e. $(\text{mult}(1,1), \delta), (\text{injive}, \delta) \in \Gamma^S$ with $\delta : (1 \overset{e}{\rightarrow} 2) \mapsto (X' \overset{e}{\rightarrow} X)$.

\[
\begin{array}{c}
X \xrightarrow{f} Y \\
\downarrow^e \\
X'
\end{array}
\]

\[
\begin{array}{c}
Y \\
\downarrow^{f'} \\
X'
\end{array}
\]

Arrows satisfying the extension properties are called extension arrows. The constraints on $e : X' \rightarrow X$ are meant to ensure that for any instance of the specification $S$, the arrow $e$ represents a total, single-valued and injective function. Therefore extension arrows can only be specified with signatures where this restriction on arrows can be expressed. For this paper we assume that our signatures always contain the predicates $[\text{mult}(n,m)]$ and $[\text{injive}]$. We use the predicate $[\text{extension}]$ as a shortcut to specify extension arrows. A constraint $([\text{extension}], \delta)$ with $\delta : (1 \overset{e}{\rightarrow} 2) \mapsto (X' \overset{e}{\rightarrow} X)$ is visualised as $X' \overset{e}{\rightarrow} X$. The corresponding constraints $([\text{mult}(1,1)], \delta)$ and $([\text{injive}], \delta)$ are not shown since they are implicitly given. Moreover, the arrows $f' : X' \rightarrow Y$ are omitted in $G^S$ because they are also implied by the establishment of $e : X' \rightarrow X$ as an extension arrow.
3. Copy-Modify-Merge Approach to Version Control in MDE

In this section the copy-modify-merge approach to version control in MDE is introduced and formalised. First we illustrate a usual scenario of concurrent development in MDE, in which an ideal copy-modify-merge VCS is adopted. In this scenario, each developer accesses a repository and creates a personal working copy – a snapshot of the repository’s models. Then, the developers modify their working copies simultaneously and independently. Finally, the working copies are merged together in the repository. The VCS assists with the merging by detecting conflicting modifications. When a conflict is detected, the system requires manual intervention of the developer.

Recall that in our formalisation, models are \( \Sigma \)-specifications as defined in Section 2. Throughout this section, the following notation has been employed:

- Model \( V_i \): a version of a model in the repository, e.g. \( V_2 \)
- Working copy \( U_i \), with \( U \) for user: a working copy of the model \( V_i \), e.g. \( A_2 \), with A for Alice.

Example 2. The following example is kept intentionally simple, retaining only the details which are relevant for our discussion. Suppose that two software developers, Alice and Bob, are working on the development of an information system for universities using a copy-modify-merge VCS. This scenario is depicted in Fig. 3 and an overview of the models in the example is shown in Fig. 4.

![Figure 3: The timeline of the example](image)

Alice creates a working copy \( A_1 \) of the model \( V_1 \) in the repository (Fig. 4a). This is done in a check-out step. She modifies her working copy by adding the node \( \text{PhDStud} \) as an extension of \( \text{Student} \), together with the arrows \( \text{PhDStud} \rightarrow \text{PhD} \). These modifications take place in an evolution step. Since other developers may have updated model \( V_1 \), she needs to synchronise her working copy with the repository in order to merge other developers’ modifications. This is done in a synchronisation step. However, no modifications of the model \( V_1 \) have been made in the repository while Alice has been modifying it. Hence, the synchronisation is completed without changing her working copy \( A_1 \). Finally, Alice commits her working copy, which will be labelled \( V_2 \) in the repository (Fig. 4b). This is done in a commit step.

Afterwards, Bob checks out a working copy \( B_2 \) of the model \( V_2 \) from the same repository. He considers \( \text{Postdoc} \) as a different type of student. To avoid the pollution of extensions in the model, he deletes the \( \text{PhDStud} \) node and refactors the model by adding two nodes, \( \text{Enrolment} \) and \( \text{Type} \), together with the arrows \( \text{Enrolment} \rightarrow \text{Type} \), \( \text{Enrolment} \rightarrow \text{PhD} \) and \( \text{Type} \rightarrow \text{PhD} \). Then, he synchronises his working copy with the repository. This synchronisation is also completed without changing his working copy \( B_2 \). Finally, Bob commits his working copy, which will be labelled \( V_3 \) in the repository (Fig. 4c).

Alice continues modifying her working copy \( A_2 \), which is now out-of-date since it is a copy of the model \( V_2 \), while the latest model in the repository (containing Bob’s modifications) is \( V_3 \). She adds...
Figure 4: The models $V_1$, $V_2$, $V_3$ and $A_2$

a node Project together with the arrows $pProjs$, $proPhds$, $proUniv$ and $uProjs$ (Fig. 4d). Then, she synchronises her working copy with the repository. This time the synchronisation procedure detects conflicting modifications. This is because Alice has added some arrows to/from the node PhDStud which Bob has deleted. The resolution of this conflict requires the manual intervention of Alice, who must review the model and decide whether to adapt it to Bob’s modifications.

The example above illustrates a usual scenario of concurrent development where conflicting modifications are detected. In the remainder of the section we analyse the underlying techniques of the copy-modify-merge approach to version control in MDE. Furthermore, we discuss the constructions adopted in our formalisation. Several examples, built on Example 2, are used to illustrate the application of our formal techniques. We extend our notation by adopting some keywords from [1].

- **Base model** $V_B$, with $B$ for BASE: the last model to be checked out or synchronised prior to any modification; i.e. the pristine version of a working copy, e.g. $V_2$ is the base model for $A_2$

- **Head model** $V_H$, with $H$ for HEAD: the latest (or most recent) model in the repository, e.g. $V_3$

Note that the head model is the same for all users. In contrast, the base model is bound to the working copy, and may differ from user to user.
3.1. Identification of Commonalities

VCSs rely on the identification of commonalities between (versions of) artefacts, which is necessary to compute their differences. For example, a solution to the longest common subsequence problem [20] is typically implemented in differencing algorithms for text-based files.

In MDE there are several different approaches to commonality identification. A rudimentary technique is based on persistent identifiers, such as Universally Unique Identifiers (UUID) [21]; in this approach model elements with equal identifiers are seen as equal elements (hard-linking) [22]. While this approach would work efficiently within specific tools, it is not general enough to function as a generic approach. This is because the generation of persistent identifiers is different for every environment. A more recent technique for the identification of commonalities is based on metrics such as structural similarity; in this approach, model elements that have the same features are seen as equal elements (soft-linking) [2]. This approach has the benefit of being general, but its current implementations are too resource greedy to be used in production environments.

In this paper, we propose a different approach to the identification of common elements. Model elements which are not changed during an evolution step are recorded in common models; i.e. models which represent the commonalities between two subsequent versions of a model. These models are regarded as meta-information about evolution steps. Common models are defined as follows (see Fig. 5):

\[ C \leftarrow \left( G^C, \Gamma^C \right), \text{ together with an injection } in_j^S : C \rightarrow S \text{ and an inclusion } inc_T : C \hookrightarrow T, \text{ is a common model of } S \text{ and } T. \]

Definition 11 (Common Model). Given models \( S \) and \( T \), a model \( C \) : \( (G^C, \Gamma^C) \), together with an injection \( in_j^S : C \rightarrow S \) and an inclusion \( inc_T : C \hookrightarrow T \), is a common model of \( S \) and \( T \).

Note that we support renaming operations by allowing arbitrary injections \( in_j^S \). We decided, however, that a common model always contains the most recent names by requiring the \( inc_T \) to be inclusions. An illustration of renaming is presented in Example 5. In our formalisation, the contribution of common models is twofold:

- For each pair of models \( V_i \) and \( V_{i+1} \), a common model \( C_{i,i+1} \) of \( V_i \) and \( V_{i+1} \) is stored in the repository. We call \( C_{i,i+1} \) the common model of \( V_i \) and \( V_{i+1} \) (Fig. 6a).

- For each pair of base model \( V_B \) and working copy \( U_B \), a local common model \( LC_B \) of \( V_B \) and \( U_B \) is recorded by the VCS. We call \( LC_B \) the common model of \( V_B \) and \( U_B \) (Fig. 6b).

![Figure 5: A common model C of the models S and T](image)

![Figure 6: Usage of common models in our formalisation](image)
Example 3. Building on Example 2, Fig. 7c shows the common model $C_{2,3}$ for the models $V_2$ and $V_3$.

![Figure 7: The common model $C_{2,3}$ of the models $V_2$ and $V_3$](image)

3.2. Calculation and Representation of Differences

As mentioned, the identification of commonalities is necessary in order to calculate the differences between artefacts. The calculation and representation of differences focuses on identifying the modifications which have taken place in each evolution step. Various approaches to difference calculation and representation in MDE can be found in the literature [2, 22, 23, 24]. These approaches differ in that they analyse the modifications which a model undergoes; e.g. change or update are given different and ambiguous semantics. Moreover, the terminology in these approaches is not precise; e.g. the terms “add”, “create” and “insert” are frequently used to refer to the same modification. In this paper, we classify modifications as indicated in Table 3. This classification is performed on the level of structural models, meaning that we do not take into account operations; i.e. methods or functions. Nor this approach does detect and represent modifications in layout or visualisation since the semantics of a structural model is not affected by these changes.

In order to calculate the difference between two subsequent versions of a model, we need to know which modifications have taken place. Identifying these modifications requires an analysis of the old and the new versions of the model, together with their common model. For example, all the nodes and arrows which are present in the new model but not in the common model are identified as added. Similarly, all the nodes and arrows which are present in the old model but not in the common model are identified as deleted. This means that in order to calculate the difference we need to distinguish between common elements, elements from the old version and elements from the new version. This capacity...
Table 3: Classification of modifications

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Alternative terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>a node/arrow is added to the underlying graph of a model</td>
<td>create, insert</td>
</tr>
<tr>
<td>delete</td>
<td>a node/arrow is deleted from the underlying graph of a model</td>
<td>remove</td>
</tr>
<tr>
<td>rename</td>
<td>a node/arrow is renamed in the underlying graph of a model</td>
<td>special case of change or update</td>
</tr>
</tbody>
</table>

is one of the properties of the pushout construction in \( \text{Spec}^\Sigma \). Hence, pushout is adopted to calculate differences between models. In particular, pushout constructs a model where all common, added, deleted and renamed elements are present at the same time.

The output of this calculation is then presented in a difference model. In order to show the modifications, an appropriate language is needed. Due to the nature of models, the language must be diagrammatic and must make it possible to identify modifications as added, deleted and renamed. In our formalisation, we use DPF to define a signature \( \Sigma_\Delta \) for the representation of model differences. The signature \( \Sigma_\Delta = (\Pi_\Delta, \alpha_\Delta) \) consists of four predicates: \([\text{add}]\), \([\text{delete}]\), \([\text{rename}(\text{old,new})]\) and \([\text{conflict}]\) (see Table 4). These predicates are used to present the information “added, deleted and renamed” locally in the difference model. In particular, the difference models will be decorated by predicates from \( \Sigma_\Delta \) in addition to predicates from \( \Sigma \). The predicates \([\text{add}]\), \([\text{delete}]\) and \([\text{rename}(\text{old,new})]\) each has two arities: 1 and \( 1 \rightarrow 2 \). That is, each of these predicates can be used to decorate both nodes and arrows. Note that the predicate \([\text{conflict}]\) is not used in the difference models of two subsequent models; it will be used to decorate conflicting modifications in the synchronisation procedure (see Section 3.3).

Table 4: The signature \( \Sigma_\Delta \)

<table>
<thead>
<tr>
<th>( [\text{add}]^a )</th>
<th>1</th>
<th>X ( ^a )</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [\text{add}]^a )</td>
<td>( 1 \rightarrow 2 )</td>
<td>X ( ^{a</td>
<td>f} )</td>
</tr>
<tr>
<td>( [\text{delete}]^a )</td>
<td>1</td>
<td>X ( ^{a</td>
<td>f} )</td>
</tr>
<tr>
<td>( [\text{delete}]^a )</td>
<td>( 1 \rightarrow 2 )</td>
<td>X ( ^{a</td>
<td>f} )</td>
</tr>
<tr>
<td>( [\text{rename}(\text{old,new})]^a )</td>
<td>1</td>
<td>Y ( [R:X \mapsto Y] )</td>
<td></td>
</tr>
<tr>
<td>( [\text{rename}(\text{old,new})]^a )</td>
<td>( 1 \rightarrow 2 )</td>
<td>X ( ^{a</td>
<td>f} )</td>
</tr>
<tr>
<td>( [\text{conflict}]^a )</td>
<td>1</td>
<td>X ( ^e )</td>
<td></td>
</tr>
<tr>
<td>( [\text{conflict}]^a )</td>
<td>( 1 \rightarrow 2 )</td>
<td>X ( ^{c</td>
<td>f} )</td>
</tr>
</tbody>
</table>

**Remark 7 (Multiple Visualisations).** We define two visualisations for the \( \Sigma_\Delta \) predicates. The default visualisation is compatible with black and white printing since predicate labels are used to decorate the model elements in difference models. Although this visualisation enables the representation of differences, an alternative visualisation based on colour-coding is proposed. We believe that this colouring technique makes it easier to understand modifications. We have adopted both visualisations so that the examples are intuitive as well as compatible with black and white printing.
In the light of the calculation and representation approaches described above, we define difference models as follows (see Fig. 8):

![Diagram](image)

**Definition 12 (Difference Model).** Given a common model $C$ of models $S$ and $T$, the difference model is a $(\Sigma \cup \Sigma_A)$-specification $D := (G^M, \Gamma^M \cup \Gamma^D_A)$, where:

1. The $\Sigma$-specification $M := (G^M, \Gamma^M)$, together with an injection $\text{inj}_M: S \to M$ and an inclusion $\text{inc}_M: T \hookrightarrow M$, is constructed as the pushout $(M, \text{inj}_M: S \to M, \text{inc}_M: T \hookrightarrow M)$ of the diagram $S \xrightarrow{\text{inj}_M} C \xleftarrow{\text{inc}_M} T$ in $\text{Spec}^2$, in accordance with Remark 5.

2. $\Gamma^D_A$ is constructed according to the following rules:
   
   (a) For each node $X \in (G^D_1 \setminus G^C_0)$:
   
   $$\text{([add]}^n, \delta) \in \Gamma^D_A \text{ where } \delta(\alpha([\text{add}]}^n)) = X$$
   
   (b) For each arrow $f \in (G^D_1 \setminus G^C_1)$:
   
   $$\text{([add]}^n, \delta) \in \Gamma^D_A \text{ where } \delta(\alpha([\text{add}]}^n)) = f$$
   
   (c) For each node $X \in (G^D_0 \setminus G^C_0)$:
   
   $$\text{([delete]}^n, \delta) \in \Gamma^D_A \text{ where } \delta(\alpha([\text{delete}]}^n)) = X$$
   
   (d) For each arrow $f \in (G^D_1 \setminus G^C_1)$:
   
   $$\text{([delete]}^n, \delta) \in \Gamma^D_A \text{ where } \delta(\alpha([\text{delete}]}^n)) = f$$
   
   (e) For each node $Y \in G^C_0$ such that $Y \neq \text{inj}_S(Y)$:
   
   $$\text{([rename}\text{(inj}_S(Y), Y)]^n, \delta) \in \Gamma^D_A \text{ where } \delta(\alpha([\text{rename}\text{(inj}_S(Y), Y)]^n)) = Y$$
   
   (f) For each arrow $g \in G^C_1$ such that $g \neq \text{inj}_S(g)$:
   
   $$\text{([rename}\text{(inj}_S(g), g)]^n, \delta) \in \Gamma^D_A \text{ where } \delta(\alpha([\text{rename}\text{(inj}_S(g), g)]^n)) = g$$

**Example 4.** Building on Example 2, Fig. 9e shows the difference model $D$ for the models $V_2$ and $V_3$. The nodes Enrollment and Type and the arrows $\text{$\&$en1}$ and $\text{$\&$Type}$ have been added to the model $V_3$. These added elements are decorated as added; i.e. decorated with the predicate $\text{[add]}$ from $\Sigma_A$ in the difference model $D$. This predicate is visualised as $[A]$ in the proposed visualisation, or by green colouring in the alternative visualisation of $\Sigma_A$.

With regard to the $\text{[rename}\text{(old, new)]}$ predicate, once a node (arrow) $X \in S$ is renamed to $Y \in T$, the common model $C$ and the difference model $D$ will contain $Y$ with $\text{inj}_S(Y) = X, \text{inc}_T(Y) = Y, \text{inj}_M(X) = Y$ and $\text{inc}_M(Y) = Y$. Moreover, the node (arrow) $Y$ will be decorated with the predicate $\text{[rename}\text{(X, Y)]}$. The morphisms $\text{inj}_S$ and $\text{inj}_M$ are injective in order to allow for this renaming. Moreover, the morphisms $\text{inc}_T$ and $\text{inc}_M$ are inclusion so that the common and the difference models always contain the new name.
Figure 9: The difference model \( D \) for the models \( V_2 \) and \( V_3 \)
Example 5. Fig. 10e shows the difference models $D$ for the models $V_0$ and $V_1$. The node $\text{Person}$ in $V_0$ is renamed to $\text{Student}$ in $V_1$. The injection $\text{inj}_{V_0}: \mathbb{C}_0 \to V_0$ contains an explicit mapping $\text{Student} \mapsto \text{Person}$; analogously, the injection $\text{inj}_{M}: V_0 \to M$ contains an explicit mapping $\text{Person} \mapsto \text{Student}$. The node $\text{Person}$ is decorated with the predicate $\text{rename(\text{Person}, \text{Student})}$ in the difference model $D$. This predicate is visualised as $\{\text{Person} \mapsto \text{Student}\}$.

![Diagram of models and difference model](image)

Figure 10: The difference model $D$ for the models $V_0$ and $V_1$ showing a renaming

3.3. Synchronisation

To enable concurrent development – which is the main goal of the copy-modify-merge approach to version control – a mechanism for model synchronisation is necessary. In this section we present a
synchronisation procedure which exploits the identification of commonalities and the calculation/representation of differences presented in the previous sections.

Whenever a working copy \(U_B\) is to be synchronised with the head model \(V_H\) from the repository, two cases are considered:

- If nobody has updated the head model \(V_H\); i.e. if the head model \(V_H\) and the base model \(V_B\) are identical, then the working copy is not affected by the synchronisation procedure.

- If someone has updated the head model \(V_H\); i.e. if the head model \(V_H\) and the base model \(V_B\) are different, then the modifications made by others will be merged into the working copy and possible conflicts will be detected.

We propose a synchronisation procedure which takes as input the following models:

- The working copy \(U_B\) and the common model \(LC_B\), which are stored locally in the snapshot of the repository.

- The head model \(V_H\), the base model \(V_B\) and their intermediate common models \(C_{B,B+1} \ldots C_{H-1,H}\), which are stored remotely in the repository.

Furthermore, the synchronisation procedure is divided into several steps. To give an impression of these steps, we have described them first informally:

1. Calculate the common of the commons for the base model and the head model.
2. Calculate the difference model for the base model and the working copy.
3. Calculate the difference model for the base model and the head model.
4. Calculate the merge of differences.
5. Calculate the processed merge of differences.
6. Detect conflicts.
7. Calculate the synchronised working copy and the synchronised common model.

Before presenting the details of our synchronisation procedure, the concepts mentioned above need to be defined. The steps above are detailed in the following:
**Step 1**

The common models in the repository represent the commonalities between subsequent versions of a model. However, we are also interested in the common model for models which are not subsequent versions of each other. This is because the synchronisation procedure will calculate the difference model of the base model $V_B$ and the head model $V_H$ which may have an arbitrary number of intermediate models $V_{B+1} \ldots V_{H-1}$ in between. We can construct this common model from the common models $C_B, B+1, \ldots, C_H$ of the intermediate models. We call this common model the *common of commons*. One possible way to compute this model is defined as follows (see Fig. 11):

$$C_i, k_{in j, k} \rightsquigarrow C_i, j_{in k, j}$$

**Definition 13 (Common of Commons).** Given models $C_{i,j}, C_{j,k}, V_i, V_j$ and $V_k$, the common of commons is a $\Sigma$-specification $C_{i,k} := (G^{\Sigma_{i,k}}, \Gamma^{\Sigma_{i,k}})$ together with the injection $f := in_{j,i}; in_{j,i}$ and the inclusion $g := inc_{j,k}; inc_{k,i}$ constructed as the pullback of $\left( \leftarrow C_{i,k} \rightarrow \leftarrow C_{i,j} \right)$ of the diagram $\rightarrow C_{j,k} \rightarrow$ of the pullback $(C_{i,k}, in_{j,i}, in_{j,i} : C_{i,k} \rightarrow C_{i,j}, inc_{j,k} : C_{i,k} \leftarrow C_{j,k})$ of the diagram $\rightarrow \leftarrow C_{i,k} \rightarrow$.

For numbers $i, k$ with $(k-i) > 2$, there are different possible ways to compute a corresponding common of commons by a sequence of pullback constructions. However, all these different sequences will produce the same result, as discussed in Remark 3. Thus, we can talk about the common of commons and use the notation $C_{i,k}$ for this.

**Remark 8 (Identities of Model Elements).** For each $i, k$ such that $i < k$, model elements which are added to $V_k$ with the same name as model elements which were deleted in $V_i$ are considered different model elements. For example, a node $\text{Student}$ which is deleted from $V_1$ and a node $\text{Student}$ which is added in $V_{10}$ are distinct nodes and they will not be identified in the common model $C_{1,10}$.

**Example 6.** Building on Example 2, Fig. 12f shows the common model $C_{1,3}$ of the common models $C_{1,2}$ and $C_{2,3}$, which is the common model for the models $V_1$ and $V_3$. 

![Figure 11: The common model $C_{i,k}$ of the common models $C_{i,j}$ and $C_{j,k}$](image-url)
Figure 12: The common model $C_{1,3}$ of the common models $C_{1,2}$ and $C_{2,3}$
Steps 2 and 3

Once the common of commons $C_{B,H}$ is available, the difference models $LD$ and $D$ of $V_B$, $U_B$ and $V_B$, $V_H$, respectively, are calculated. These difference models are used to synchronise $U_B$ with $V_H$. More precisely, the difference models $LD$ and $D$ are merged in the merge of differences $MD$. This model is then processed in order to detect conflicts which are presented in the processed merge of differences $PM$.

Step 4

The definition of merge of differences is as follows (see Fig. 13):

\[
\begin{array}{ccc}
LD & \xrightarrow{lin_{MD}} & V_B \\
\downarrow{lin_{MD}} & & \downarrow{in_{MD}} \\
MD & \xrightarrow{in_{MD}} & D \\
\end{array}
\]

Figure 13: The merge of differences $MD$ and the processed merge of differences $PM$

**Definition 14 (Merge of Differences).** Given models $LD$, $D$ and $V_B$, the merge of differences is a $(\Sigma \cup \Sigma_\Delta)$-specification $MD := (G^{MD}, \Gamma^{MD} \cup \Gamma^{MD}_\Delta)$ together with the injections $lin_{MD} : LD \to MD$ and $in_{MD} : D \to MD$ constructed as the pushout $(MD, lin_{MD} : LD \to MD, in_{MD} : D \to MD)$ of the diagram $LD \xrightarrow{lin_{MD}} V_B \xleftarrow{in_{MD}} D$ in $Spec^\Sigma$, in accordance with Proposition 2.

The sets of constraints $\Gamma^LD_\Delta$ and $\Gamma^D_\Delta$ are merged together into $\Gamma^MD_\Delta$. While some of these constraints are identified by the pushout construction (see Remark 6), some model elements may be decorated by two predicates from $\Sigma_\Delta$. Each pair of constraints in $\Gamma^MD_\Delta$ must be analysed to detect conflicting modifications; i.e. to add constraints $\{\text{[conflict]}, \delta\} \in \Gamma^MD_\Delta$. In order to perform this analysis, we have defined a series of rules in Definition 15 which are applied to $MD$. These rules contain all possible combinations of constraints $(p, \delta) \in \Gamma^MD_\Delta$. This is justified as follows:

- It is impossible to have the constraint $\{\text{[add]}, \delta\}$ together with any other constraint on the same model element in $MD$. This is because we consider elements added by different users to be distinct even if they have the same names, and only elements which are identified in a common model will be treated as identical.

- It is impossible to have the constraints $\{\text{[delete]}, \delta\}$ twice on the same model element in $MD$. This is because the constraints are identified by the pushout construction.

<table>
<thead>
<tr>
<th>$LD$</th>
<th>$D$</th>
<th>${\text{add}}$</th>
<th>${\text{delete}}$</th>
<th>${\text{rename(\text{old,new})}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[add]</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Impossible</td>
<td></td>
</tr>
<tr>
<td>[delete]</td>
<td>Impossible</td>
<td>Identified</td>
<td>Possible</td>
<td></td>
</tr>
<tr>
<td>[rename(\text{old,new})]</td>
<td>Impossible</td>
<td>Possible</td>
<td>Possible</td>
<td></td>
</tr>
</tbody>
</table>
Should the processed merge of differences $PM$ contain a constraint ($\texttt{[conflict]},\delta$), the working copy will be in a state of conflict which has to be resolved manually by the developer. Although conflicts are context-dependent, we have recognised some situations in which syntactic conflicts will arise. In addition, rules for the detection of custom conflicting modifications can be defined. The following is a summary of the concurrent modifications which we identify as conflicts:

- Renaming an element which has been deleted.
- Renaming an element which has been renamed.
- Adding arrows from/to a node which has been deleted.

Steps 5 and 6

The definition of the processed merge of differences is as follows (see Fig. 13):

**Definition 15 (Processed Merge of Differences).** Given a model $MD$, the processed merge of differences is a $(\Sigma \cup \Sigma{\Delta})$-specification $PM := (G^{PM},\Gamma{\Delta})$ together with the injection $p_{PM}: PM \rightarrow MD$, where:

1. We set $G^{PM} := G^{MD}$ and $p_{PM}: PM \rightarrow MD$ is given by the identity on $G^{PM} = G^{MD}$.
2. $\Gamma{\Delta}$ is constructed and $G^{PM}, p_{PM}$ are changed by the application of the following rules (see Table 5):

   - For each node (arrow) $X \in G^{MD}$, we have:
     
     (a) $([\texttt{rename}(X,Y)],\delta;lin_{MD}) \in \Gamma{\Delta}$ implies $X \in G^{PM}$ is replaced by $Y$, we set $p_{PM}(Y) = X$ and $([\texttt{rename}(X,Y)],\delta) \in \Gamma{PM}$ where $\delta(\alpha([\texttt{rename}(X,Y)]) = Y$.

     $([\texttt{rename}(X,Y)],\sigma;in_{MD}) \in \Gamma{\Delta}$ implies $X \in G^{PM}$ is replaced by $Y$, we set $p_{PM}(Y) = X$ and $([\texttt{rename}(X,Y)],\delta) \in \Gamma{PM}$ where $\delta(\alpha([\texttt{rename}(X,Y)]) = Y$.

   - For each node (arrow) $X \in G^{MD}$ and pair of constraints $(p,\delta;lin_{MD}), (q,\sigma;in_{MD}) \in \Gamma{\Delta}$ such that $X = (\delta;lin_{MD})(a(p)) = (\sigma;in_{MD})(a(q))$, we have:

     (b) $([\texttt{delete}],\delta;lin_{MD}), ([\texttt{rename}(X,Y)],\sigma;in_{MD}) \in \Gamma{\Delta}$ implies $([\texttt{delete}],\delta;lin_{MD}), ([\texttt{rename}(X,Y)],\sigma;in_{MD}), ([\texttt{conflict}],\delta) \in \Gamma{PM}$.

     (c) $([\texttt{rename}(X,Y)],\delta;lin_{MD}), ([\texttt{rename}(X,Z)],\sigma;in_{MD}) \in \Gamma{\Delta}$ implies $([\texttt{rename}(X,Y)],\delta;lin_{MD}), ([\texttt{rename}(X,Z)],\sigma;in_{MD}), ([\texttt{conflict}],\delta) \in \Gamma{PM}.

     (d) $([\texttt{add}]^a,\sigma;in_{MD}), ([\texttt{delete}]^b,\sigma;in_{MD}) \in \Gamma{\Delta}$ implies $([\texttt{add}]^a,\delta;lin_{MD}), ([\texttt{delete}]^b,\sigma;in_{MD}), ([\texttt{conflict}]^c,\delta;lin_{MD}), ([\texttt{conflict}]^d,\sigma;in_{MD}) \in \Gamma{\Delta}$. 


Table 5: A summary of the reduction rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>MD</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td><img src="MD1.png" alt="Diagram" /></td>
<td><img src="PM1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>(b)</td>
<td><img src="MD2.png" alt="Diagram" /></td>
<td><img src="PM2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>(c)</td>
<td><img src="MD3.png" alt="Diagram" /></td>
<td><img src="PM3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>(d)</td>
<td><img src="MD4.png" alt="Diagram" /></td>
<td><img src="PM4.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

In short, the purpose of the rules above is as follows:

(a) Applies the renaming of model elements.
(b) Detects a conflict in case a model element is renamed and deleted concurrently.
(c) Detects a conflict in case a model element is renamed twice concurrently.
(d) Detects a conflict in case a node is deleted while an arrow having source or target to this node is added (dangling edges).

**Example 7.** Building on Example 2, Fig. 14j shows the merge of differences MD for the difference models LD and D, while Fig. 14k shows the processed merge of differences PM. Note that the arrows pUnivs and pDiffs are decorated with the predicate [delete] in both LD and D, however, since the two corresponding constraints are identified by the pushout construction when MD is created, there is only one [delete] in MD. Moreover, the node pDiffs and the arrows pProjs and pDiffs are decorated with [delete], [add] and [add] in MD, respectively. In PM these nodes and arrows will be additionally decorated with [conflicts] in accordance with Definition 15, Rule (d).

**Step 7**

Synchronised working copy and synchronised common models are defined as follows (see Fig. 16):

**Definition 16 (Synchronised Working Copy).** Given a model PM, the synchronised working copy is a Σ-specification \( U_H := (G_{U_H}^{PM}, \Gamma_{U_H}) \) together with the inclusion \( s_{U_H} : U_H \hookrightarrow \) PM where:

1. \( G_{U_H}^{PM} := G_0^{PM} \setminus \{ \delta(\text{[delete]}^a) \} \) \( \cap \{ \text{[delete]}^a, \delta \in \Gamma_{PM} \} \)
2. \( G_{U_H}^{PM} := G_1^{PM} \setminus \{ \delta(\text{[delete]}^a) \} \) \( \cap \{ \text{[delete]}^a, \delta \in \Gamma_{PM} \} \)
3. \( \text{src}^{G_{U_H}}(f) = \text{src}^{G_{U_H}}(f) \) and \( \text{trg}^{G_{U_H}}(f) = \text{trg}^{G_{U_H}}(f) \) for all \( f \in G_{U_H}^{PM} \)
4. For each constraint \( (p, \delta) \in \Gamma^{PM} \) such that \( \delta(\alpha(p)) \subseteq G_{U_H}^{PM}, (p, \delta) \in \Gamma^{U_H} \).

**Definition 17 (Synchronised Common Model).** Given models \( V_H \) and \( U_H \), the synchronised common model is a Σ-specification \( L_{CH} := (G_{LCH}^{PM}, \Gamma_{LCH}^{PM}) \) together with the injection \( linj_{V_H} : L_{CH} \rightarrow V_H \) and the inclusion \( inc_{U_H} : L_{CH} \hookrightarrow U_H \), constructed as the pullback \( (L_{CH}, \text{linj}_{V_H}) : L_{CH} \rightarrow V_H, \text{inc}_{U_H} : L_{CH} \hookrightarrow U_H \) of the diagram \( V_H \xrightarrow{inc_{U_H}[inc_{U_H} \circ \text{linj}_{V_H}]} MD \xrightarrow{\text{linj}_{V_H} \circ inc_{U_H}} U_H \) in Spec Σ, in accordance with Proposition 1.
Example 8. Building on Example 2 once again, Fig. 15l shows how the synchronised working copy \( A_3 \) would appear if constructed from the conflicting processed merge of differences \( PM \). The underlying graph of \( A_3 \) is invalid since it contains “dangling edges”. Note that this model will not be constructed by the synchronisation procedure. That is, the presence of \( ([\text{conflict}], \delta) \) constraints in \( \Gamma^\Delta_{A_3} \) prevents the synchronisation procedure from creating \( A_3 \).
The next procedure explains the details of our approach to synchronisation (see Fig. 16):

Procedure 1 (sync procedure).

1. if $V_B < V_H$
2.   given $V_B$, $U_B$ and $LC_B$, compute the difference model $LD$;
3. else
4.   the common model $C_{B,H}$ is given;
5.   given $V_B$, $V_H$ and $C_{B,H}$, compute the difference model $D$;
6. else
7.   given $V_B$, $LD$ and $D$ compute the merge of the differences $MD$;
8.   given $MD$ compute the processed merge of differences $PM$;
9. if $\{\text{conflict}\,...,\delta\} \notin \Gamma^A$
10.   given $PM$ compute the synchronised working copy $U_H$ and the synchronised common model $LC_H$;
11. else
12.   display $PM$;
13.   ask for manual resolution of the conflict;
Finally, when all the building blocks for synchronising the working copy with the head model are in place, as detailed in the previous seven steps, the synchronisation operation can be fulfilled. This operation is defined as follows (see Fig. 16):

**Definition 18 (Synchronisation).** Given \( U_B, V_B, V_H, LC_B, C_{B:B+1} \ldots C_{B:H-1}, C_B, B+1 \ldots C_{H-1} H \), the synchronisation \( \text{sync} : (U_B, V_B, V_H, LC_B, C_{B:B+1} \ldots C_{B:H-1}) \mapsto (U_H, LC_H, \text{inc}_U : LC_H \hookrightarrow U_H, \text{lin}_{V_H} : LC_H \rightarrow V_H) \) is a procedure which generates a synchronised working copy \( U_H \) and a synchronised common model \( LC_H \), in accordance with Procedure 1.

Once the synchronisation operation is performed without detecting any conflicts, the synchronised working copy may be committed to the repository. The committed model will be the new head model, labelled \( V_H+1 \) in the repository. In addition, the commit operation will add the synchronised common model as the common model of \( V_H \) and \( V_H+1 \). The committed common model will be labelled \( C_{H:H+1} \) in the repository. Formally, the commit operation is defined as follows (see Fig. 16):

**Definition 19 (Commit).** Given a synchronisation \( \text{sync} : (U_B, V_B, V_H, LC_B, C_{B:B+1} \ldots C_{B:H-1}) \rightarrow (U_H, LC_H, \text{inc}_U : LC_H \hookrightarrow U_H, \text{lin}_{V_H} : LC_H \rightarrow V_H) \), the commit \( \text{commit} : (U_H, LC_H, \text{inc}_U, \text{lin}_{V_H}) \rightarrow (V_H+1, \text{inc}_{V_H+1} : C_{H:H+1} \hookrightarrow V_H+1, \text{lin}_{V_H} : C_{H:H+1} \rightarrow V_H) \) is an operation which adds the models \( U_H \) and \( LC_H \) to the repository as \( V_H+1 \) and \( LC_H \), respectively, and the morphisms \( \text{inc}_U, \text{lin}_{V_H} \) as \( \text{inc}_{V_H+1}, \text{lin}_{V_H+1} \), respectively.

In the previous sections, we have illustrated our formalisation of the copy-modify-merge approach by using a running example. In the following, we have revised the example and shown all the above mentioned steps in one. Note that the synchronisation procedure in the example fails because it detects a conflict.

**Example 9.** A complete execution of the synchronisation procedure for Example 2 is shown in Fig. 17. Once a conflict is detected, the synchronisation stops at the calculation of \( PM \).
Figure 17: The complete execution of the synchronisation procedure
4. Related Work

The literature on model evolution is abundant. Firstly, there is the issue of model differencing; DSM-Diff [2] and EMF Compare [25] are two model differencing tools which are based on a similar technique. Difference calculation is divided in two phases. The first focuses on model mappings, where all the elements of the two input models are compared using measures like signature matching and structural similarity. The second phase determines model differences, detecting all the additions, deletions and changes. The great benefit of this approach is that it is general, but this is at the price of being resource greedy.

Compared to this approach, our calculation of model differences should require less resources since an explicit representation of commonalities between models is recorded and stored in common models. This avoids the need for structural similarity comparisons.

Secondly, there is the issue of how to represent differences among models that conform to an arbitrary metamodel. Typical approaches represent model differences as follows:

– As models which conform to a difference metamodel. The difference metamodel can be generic [23], or obtained by an automated transformation [3]. These models are in general minimalistic (i.e. only the necessary information to represent the difference is presented), transformative (i.e. each difference model induces a transformation), compositional (i.e. difference models can be composed sequentially or in parallel) and typically symmetric (i.e. the inverse of a given difference representation can be computed).

– As a model which is the union of the two compared models, with the modified elements highlighted by colours, tags, or symbols [22]. The adoption of this technique is typically beneficial for the designer, since the rationale of the modifications is easily readable. However, these benefits apply only if the base models are not large and not too many updates apply to the same elements, since the difference model resorts to both base models to denote the differences.

– As a sequence of atomic actions specifying how the initial model has been procedurally modified [26]. While this technique has the great advantage of being efficient, difference representation is neither readable nor intuitive. In addition, edit scripts do not follow the “everything is a model vision” [27]. They are suitable for internal representations but quite ineffective for documenting modifications in MDE environments.

According to this classification, our representation of model differences falls into the second category. We represent differences by showing the union of the two compared models and tag the modified elements with predicates (which are also highlighted by colours to enhance readability).

Thirdly, there is research focusing on 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, on the basis that it is not possible to provide a generic technique for conflict detection with arbitrary accuracy. However, in [24] 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. Currently, our formalisation enables the detection of only syntactic conflicts. However, we intend to analyse predicate dependencies and define a logic for that in a future work. This will enable us to explore the dimension of semantic conflicts as well.

A fourth strand of research focuses on the problem of heterogeneous synchronisation. In [28] the authors propose a tutorial which aims at exploring the design space of heterogeneous synchronisation. The term heterogeneous synchronisers is used by the authors to denote procedures that automate – fully or in part – the synchronisation process for (software) artefacts which are expressed in different languages. Various approaches to synchronisation of heterogeneous software artefacts are analysed and compared. In particular, the tutorial covers both the simpler synchronisation scenarios where some artefacts are never
edited directly but are re-generated from other artefacts, and the more complex scenarios where several artefacts that can be modified directly need to be synchronised.

Heterogeneous modelling languages and metamodelling are important dimensions of MDE. However, in the present paper we have not fully explored these dimensions. Our synchronisation procedure takes as input homogeneous models expressed in one modelling language. The aim of our formalisation has been to cover all aspects of the copy-modify-merge approach to version control, and provide formal definitions of these aspects in terms of category-theoretical constructs.

Finally, in the field of relational databases, category-theoretical constructs have been applied to formalise the so-called view update problem [29]; i.e. given an update to the state of a view of a database, determine an appropriate update to the state of the total database. In a future work we will compare our synchronisation procedure with the approach to view updates given in [29].

5. Conclusion and Future Work

In this paper, key aspects of the copy-modify-merge approach to version control in MDE have been analysed, formalised and illustrated. The concepts of identification of commonalities and calculation of differences have been introduced and defined as pullback and pushout, respectively. In our formalisation, we have used DPF to define a signature $\Sigma$ which is used to present model differences. The predicates of $\Sigma$ enable us to present the modifications a model may undergo; that is, to decorate model elements which have been added, deleted or renamed by predicates from $\Sigma$. In addition, we have described how these predicates can be used for the identification of possible conflicting modifications and analysed how operations such as synchronise and commit perform in distributed development. Finally, we have presented a synchronisation procedure which is based on our approach to identification of commonalities and calculation of differences.

The synchronisation procedure is described in the following steps. Firstly, we calculate the common of the commons for the base model and the head model. This step is necessary only if the head model is not a direct subsequent of the base model. Secondly, we calculate the difference model for the base model and the working copy. Then, we calculate the difference model for the base model and the head model. Next, we merge these difference models and process the merge of the differences based on a set of rules. These rules are used to apply renaming of model elements and to detect various kinds of conflicting modifications. As a final step, if no conflicts are detected a synchronised working copy and a synchronised common model are calculated. These synchronised models are stored in the repository by a commit operation.

In this paper, we have focused only on the detection of a predefined set of syntactic conflicts which are derived from experience. In a future work, we will analyse and formalise semantic conflicts; i.e. modifications which violate the constraints given by the modelling language. Moreover, we plan to introduce a reasoning system for the analysis of predicate dependencies and to define a logic for this analysis.

The formalisation given in this paper ensures the necessary foundation for tool support for version control in MDE. However, this is a challenging task, given the lack of mature standards for model serialisation and the issues related to the identification of model elements [30]. A prototype tool based on the EMF platform [31] is currently being developed. The implementation of these techniques will allow us to compare our approach with existing approaches and to carry out a full-size case study.
References


