Handling constraints in model versioning

Alessandro Rossini
PwC Norway
alessandro.rossini@pwc.com

Adrian Rutle
Western Norway University of Applied Sciences
aru@hvl.no

Yngve Lamo
Western Norway University of Applied Sciences
yla@hvl.no

Uwe Wolter
University of Bergen, Norway
wolter@ii.uib.no

ABSTRACT

In model-driven software engineering (MDSE), models are first-class entities of software development and undergo a complex evolution during their life-cycles. As a consequence, there is a growing need for techniques and tools to support model management activities such as versioning. Traditional versioning systems target text-based artefacts and are not suitable for graph-based structures such as software models. To cope with this problem, a few prototype model versioning systems have been developed. However, a uniform formalisation of model merging, conflict detection and conflict resolution in MDSE is still debated in the literature. In this paper, we propose an approach to constraint-aware model versioning; i.e., an approach that handles constraints in model merging, conflict detection and conflict resolution. The proposed approach is based on the Diagram Predicate Framework (DPF), which is founded on category theory and graph transformation.

KEYWORDS

Model-Driven Software Engineering, Model Versioning, Constraints, Category Theory, Graph Transformation, Diagram Predicate Framework

1 INTRODUCTION

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 use of modelling languages in software development processes. Software models are abstract representations of software systems that are used to tackle the complexity of present-day software by enabling developers to reason at a higher level of abstraction.

In model-driven software engineering (MDSE), models are first-class entities of software development and undergo a complex evolution during their life-cycles. As a consequence, there is a growing need for techniques and tools to support model management activities such as versioning.

In *optimistic* versioning, each developer has a local (or working) copy of a software artefact. These local copies are modified independently and in parallel. From time to time, local modifications are merged. In the *centralised* approach to optimistic versioning, local modifications from each developer are merged into a central repository. In the *distributed* approach, in contrast, local modifications from each developer are merged into other developers' local copies. In both cases, the merge is performed using a *three-way* merging technique [29], which attempts to merge two versions of a software artefact relying on the common ancestor version from which both versions originated. This technique facilitates conflict detection. In general, conflicts may arise when the modifications are contradictory. They are resolved either manually or—where applicable—automatically.

Mainstream versioning systems, e.g., Subversion [5] and Git [24], target text-based artefacts. The underlying techniques such as merging, conflict detection and conflict resolution are based on a per-line textual comparison [25]. Since the underlying structure of models is graph-based rather than text-based, these techniques are not suitable for such models [1, 2].

To cope with this problem, a few prototype model versioning systems have been developed, e.g., [3, 8]. However, a uniform formalisation of model merging, conflict detection and conflict resolution in MDSE is still debated in the literature. Research has lead to a number of findings in this field [4, 7]. The interested reader may consult [9–11, 13, 35, 41, 43] for different approaches to model merging, conflict detection and conflict resolution. Unfortunately, these techniques consider only model elements and their conformance to the corresponding modelling language, e.g., well-formedness constraints. However, these techniques should also consider constraints added to model elements, e.g., multiplicity constraints. Therefore, an interesting challenge is to extend the current techniques by enabling versioning of constraints.

In this paper, we describe a formal approach to constraint-aware model versioning based on the Diagram Predicate Framework (DPF) [33, 36]; i.e., a formal approach to model versioning that handles constraints in model merging, conflict detection and conflict resolution. In particular, the proposed approach enables the detection, and—where applicable—resolution of syntactic and semantic conflicts on constraints. This paper further develops the formalisation of model versioning published in [35, 37]. Compared to the previous work, the theoretical foundation and the underlying techniques are updated to handle constraints. Moreover, new examples are added to illustrate how model merging, conflict detection and conflict resolution are adapted to handle constraints.

The remainder of the paper is structured as follows. Section 2 introduces constraint-aware model versioning through a running example. Section 3 outlines DPF. Section 4 discusses the proposed approach to constraint-aware model versioning, with a particular focus on model merging, conflict detection and conflict resolution for constraints in DPF. In Section 5, the current research in model versioning is summarised. Finally, in Section 6, some concluding remarks and ideas for future work are presented.

2 MODEL VERSIONING

This section introduces constraint-aware model versioning. The following example illustrates a common scenario of concurrent development in MDSE. The example is kept simple intentionally, retaining only the details that are relevant for the discussion.

Example 2.1 (Model versioning and conflict detection scenario). Suppose that two developers, Alice and Bob, adopt an optimistic, centralised versioning system to develop an information system for the management of students and universities. Fig. 1 illustrates the interaction between Alice, Bob, and the repository. Fig. 2 shows the different versions of the model being developed.

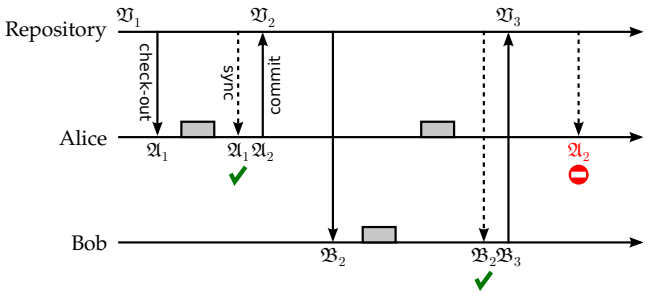


Figure 1: The timeline of the scenario

Alice creates a local copy $\mathfrak{A}_1$ of the model $\mathfrak{B}_1$ in the repository (see Fig. 2(a)). This is done in a *check-out* step. She modifies her local copy by adding the node University together with the arrows sUnivs and uStuds. These modifications take place in an *evolution* step. Since other developers may have updated the model $\mathfrak{B}_1$, she needs to synchronise her local copy with the repository to merge other developers' modifications. This is done in a *synchronisation* step. However, no modifications of the model $\mathfrak{B}_1$ have been made in the repository while Alice has been modifying it. Hence, the synchronisation is completed without changing her local copy $\mathfrak{A}_1$. Finally, Alice commits her local copy, which is labelled $\mathfrak{B}_2$ in the repository (see Fig. 2(b)). This is done in a *commit* step.

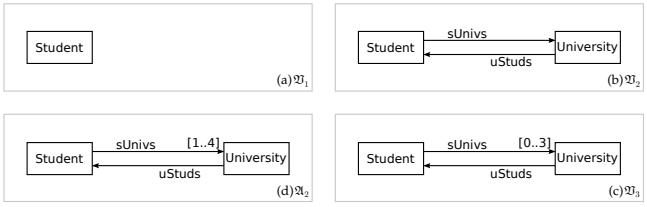


Figure 2: The models $\mathfrak{B}_1$, $\mathfrak{B}_2$, $\mathfrak{B}_3$ and $\mathfrak{A}_2$

Afterwards, Bob checks out a local copy $\mathfrak{B}_2$ of the model $\mathfrak{B}_2$ from the same repository. He adds the multiplicity constraint [0..3] on the arrow sUnivs. Then, he synchronises his local copy with the repository. This synchronisation is also completed without changing his local copy $\mathfrak{B}_2$. Finally, Bob commits his local copy, which is labelled $\mathfrak{B}_3$ in the repository (see Fig. 2(c)).

Alice continues to modify her local copy $\mathfrak{A}_2$, which is now *out-of-date* since it is a copy of the model $\mathfrak{B}_2$, while the head model in the repository (containing Bob's modifications) is $\mathfrak{B}_3$. She adds the multiplicity constraint [1..4] on the same arrow sUnivs. Then, she synchronises her local copy with the repository. The synchronisation procedure detects conflicting modifications. This is because Alice has added a multiplicity constraint that contradicts the one added by Bob.

3 DIAGRAM PREDICATE FRAMEWORK

DPF is a formal diagrammatic specification framework founded on category theory and graph transformation. The interested reader may consult [33–36, 38] for a more detailed presentation of the framework. DPF is an extension of the Generalized Sketches Framework originally developed by Diskin et al. in [14–20]. This section presents the basic concepts of DPF that are used in the formalisation of constraint-aware model versioning.

In DPF, a model is represented by a *specification* $\mathfrak{S}$. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of an *underlying graph* S together with a set of *atomic constraints* $C^{\mathfrak{S}}$ that are specified by means of a *predicate signature* Σ. A predicate signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of *predicates* $\pi \in \Pi^{\Sigma}$, each having an arity (or shape graph) $\alpha^{\Sigma}(\pi)$. An atomic constraint (π, δ) consists of a predicate $\pi \in \Pi^{\Sigma}$ together with a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$ from the arity of the predicate to the underlying graph of a specification.

Example 3.1 (Signature and specification). Building on Example 2.1, Table 1 shows a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ suitable for object-oriented structural modelling. The first column of the table shows the predicate symbols. The second and the third columns show the arities of predicates and a proposed visualisation of the corresponding atomic constraints, respectively. Finally, the fourth column presents the semantic interpretation of each predicate.

Table 1: The signature Σ

$\pi \in \Pi^{\Sigma}$	$\alpha^{\Sigma}(\pi)$	Proposed vis.	Semantic interpretation
[mult(m, n)]	$1 \xrightarrow{a} 2$	X —f→ Y, [m..n]	$\forall x \in X : m \leq \lvert f(x)\rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
[surjective]	$1 \xrightarrow{a} 2$	X —f→ Y, [surj]	$\forall y \in Y \; \exists x \in X : y \in f(x)$
[inverse]	$1 \underset{b}{\overset{a}{\rightleftarrows}} 2$	X —f→ Y, [inv], g	$\forall x \in X, \forall y \in Y : y \in f(x)$ if and only if $x \in g(y)$

Fig. 3(a) shows a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ representing an object-oriented structural model. Fig. 3(b) shows the underlying graph S of the specification $\mathfrak{S}$, i.e., the graph of $\mathfrak{S}$ without any atomic constraints.

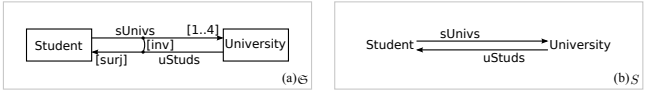


Figure 3: A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ and its underlying graph S

In $\mathfrak{S}$, the nodes Student and University are interpreted as sets $Student$ and $University$, while the arrows sUnivs, uStuds are interpreted as multi-valued functions $sUnivs : Student \to \wp(University)$, etc., where the powerset $\wp(University)$ is the set of all subsets of $University$, i.e., $\wp(University) = \{A \mid A \subseteq University\}$.

The function $sUnivs$ has cardinality between one and four. In $\mathfrak{S}$, this is enforced by the atomic constraint $([\texttt{mult}(1,4)], \delta_1)$ on the arrow sUnivs. This atomic constraint is formulated by the predicate $[\texttt{mult}(m,n)]$ from the signature Σ (see Table 1). Moreover, the function $uStuds$ is *surjective*. In $\mathfrak{S}$, this is enforced by the atomic constraint $([\texttt{surjective}], \delta_3)$ on the arrow uStuds. Furthermore, the functions $sUnivs$ and $uStuds$ are *inverse* of each other; i.e., $\forall s \in Student$ and $\forall u \in University : s \in uStuds(u)$ if and only if $u \in sUnivs(s)$. In $\mathfrak{S}$, this is enforced by the atomic constraint $([\texttt{inverse}], \delta_2)$ on the arrows sUnivs and uStuds.

The semantics of predicates of the signature Σ (see Table 1) is described using the mathematical language of set theory. In an implementation, the semantics of a predicate is typically given by the code of a corresponding validator such that the mathematical and the validator semantics coincide [28].

A semantic interpretation $[\![..]\!]^\Sigma$ of a signature Σ consists of a mapping that assigns to each predicate symbol $\pi \in \Pi^\Sigma$ a set $[\![\pi]\!]^\Sigma$ of graph homomorphisms $\iota : O \to \alpha^\Sigma(\pi)$, called valid instances of π, where O may vary over all graphs. $[\![\pi]\!]^\Sigma$ is assumed to be closed under isomorphisms. The interested reader may consult [36, 38] for a more detailed discussion of the semantics of a specification.

Example 3.2 (Instance of a specification). Building on Example 3.1, Fig. 4(a) shows a valid instance of $\mathfrak{S}$, while Fig. 4(b) shows an invalid instance of $\mathfrak{S}$ that violates the atomic constraints $([\texttt{surjective}], \delta_3)$ and $([\texttt{inverse}], \delta_2)$ since Adrian is associated with four universities but none of these universities are associated with Adrian.

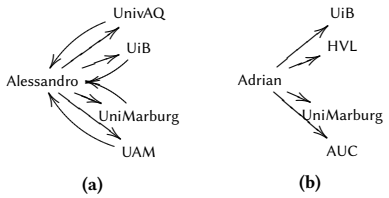


Figure 4: (a) A valid instance of $\mathfrak{S}$; (b) An invalid instance of $\mathfrak{S}$ violating $([\texttt{surjective}], \delta_3)$ and $([\texttt{inverse}], \delta_2)$

4 CONSTRAINT-AWARE MODEL VERSIONING

This section introduces the underlying techniques of the proposed approach to constraint-aware model versioning, with a particular focus on model merging, conflict detection and conflict resolution for atomic constraints. The interested reader may consult [33, 35] for a more detailed discussion of model versioning in DPF.

4.1 Representation of differences

In this paper, the difference between two versions of a specification is represented by a *difference specification*; i.e., a specification that contains all common, added, deleted and renamed elements. The motivation behind adopting difference specifications is that—as will be clear later—gathering all these elements in one specification facilitates the application of transformation rules to detect and resolve conflicts.

Due to the diagrammatic nature of specifications, the *representation* of differences such as added, deleted and renamed is expressed by a diagrammatic language. The diagrammatic language for the representation of differences is given by a *label signature* Δ, which has the same structure of a signature but no semantic counterpart (see Table 2). A label signature $\Delta = (\Theta^\Delta, \alpha^\Delta)$ consists of a set of *labels* $\theta \in \Theta^\Delta$, each having an arity $\alpha^\Delta(\theta)$. Hence, a difference specification $\mathfrak{D} = (D, C^\mathfrak{D} : \Sigma, A^\mathfrak{D} : \Delta)$ consists of a specification $\mathfrak{D}$ together with a set of annotations $A^\mathfrak{D}$ that are specified by means of the label signature Δ (see, e.g., $\mathfrak{U}D$ and $\mathfrak{D}$ in Fig. 5).

Table 2: A subset of the signature Δ for the annotation of atomic constraints

$\theta \in \Theta^\Sigma$	$\alpha^\Delta(\theta)$	Proposed visual.
<add>$^{[\texttt{mult}(m,n)]}$	$1 \xrightarrow{a} 2$	X —f, ++[m..n]→ Y
<delete>$^{[\texttt{mult}(m,n)]}$	$1 \xrightarrow{a} 2$	X —f, - -[m..n]→ Y
<conflict>$^{[\texttt{mult}(m,n)]}$	$1 \xrightarrow{a} 2$	X —f, [m..n]→ Y

The underlying graph and the set of atomic constraints of a difference specification are obtained by the pushout construction [6, 23, 33], which can be regarded as a generalisation of union. In this paper, we do not provide details of the pushout construction, but we offer an explanation of its usage by means of the running example.

Example 4.1 (Difference specification). Building on Example 2.1, recall that two (or more) developers may modify the same specifications concurrently. Starting from the base specification $\mathfrak{B}_2$ (see Fig. 2(b)), Bob makes and commits his modifications so that the head specification is now $\mathfrak{B}_3$ (see Fig. 2(c)). Concurrently, Alice makes (but does not commit) her modifications so that her local copy is now $\mathfrak{A}_2$ (see Fig. 2(d)). Next, Alice synchronises her local copy with the head specification.

The specification $\mathfrak{U}\mathfrak{D}$ (see Fig. 5(g)) represents the difference between Alice's local copy and the base specification. It is calculated using the pushout construction by merging $\mathfrak{A}_2$ and $\mathfrak{B}_2$ modulo their commonality specification, which contains the unmodified elements from $\mathfrak{B}_2$ to $\mathfrak{A}_2$.

Similarly, the specification $\mathfrak{D}$ (see Fig. 5(d)) represents the difference between the head specification and the base specification. It is calculated using the same pushout construction by merging $\mathfrak{B}_3$ and $\mathfrak{B}_2$ modulo their commonality specification, which contains the unmodified elements from $\mathfrak{B}_2$ to $\mathfrak{B}_3$.

4.2 Conflict detection

The merge of differences $\mathfrak{MD}$ is a specification that represents the concurrent modifications of two (or more) developers and that is processed to detect conflicts. Conflicts are specified by *conflict detection rules*, which are based on *constraint-aware transformation rules* [33, 36, 38].

A transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ consists of three specifications $\mathfrak{L}$, $\mathfrak{K}$ and $\mathfrak{R}$. $\mathfrak{L}$ and $\mathfrak{R}$ are the *left-hand side* and *right-hand side* of the transformation rule, respectively, while $\mathfrak{K}$ is their interface. $\mathfrak{L} \setminus l(\mathfrak{K})$ describes the part of a specification that is to be deleted, $\mathfrak{R} \setminus \mathfrak{K}$ describes the part to be added, and $\mathfrak{K}$ describes the part that has to exist to apply the rule, in which only renaming modifications are possible. Note that the specification morphism $l : \mathfrak{K} \to \mathfrak{L}$ is injective—not an inclusion—to allow for renaming [33].

An *application of transformation rule* consists of finding a match for the left-hand side $\mathfrak{L}$ in a source specification $\mathfrak{S}$ and replacing $\mathfrak{L}$ with $\mathfrak{R}$, leading to a target specification $\mathfrak{S}'$.

A conflict detection rule consists of a non-deleting transformation rule, where the left-hand side $\mathfrak{L}$ represents the conflict and the right-hand side $\mathfrak{R}$ is a specification where the conflicting elements are annotated. The interface $\mathfrak{K}$ is equal to $\mathfrak{L}$ since non-deleting transformation rules do not delete any elements.

Detecting a conflict consists of applying a conflict detection rule by finding a match for the left-hand side $\mathfrak{L}$ in the merge of differences $\mathfrak{MD}$, leading to a target merge of differences $\mathfrak{MD}''$ where the conflicting elements are annotated.[1] Hence, $\mathfrak{MD}$ is processed by applying all applicable conflict detection rules.

Applying these rules provides a classification of the conflicts into two kinds, namely *traditional conflict* and *custom conflict*. A traditional conflict occurs when concurrent modifications compete over the same elements of a specification; e.g., a part of the underlying graph is deleted while an atomic constraint having the same part as the target is added (dangling atomic constraints). Table 3 (rule g) shows a traditional conflict detection rule for dangling multiplicity constraints. In contrast, a custom conflict occurs when concurrent modifications lead to contradicting constraints; e.g., two (different) multiplicity constraints are added to the same arrow. Table 3 (rule l) shows a custom conflict detection rule for inconsistent multiplicity constraints. Note that these kinds of conflicts may include domain-specific conflicts. The proposed approach enables the specification of custom conflict detection rules on demand.

The following example illustrates the application of custom conflict detection rules.

Example 4.2 (Merge of differences and custom conflict detection). Building on Example 2.1, Fig. 5(h) shows the merge of differences $\mathfrak{MD}$. It is calculated using the pushout construction on $\mathfrak{UD}$, $\mathfrak{D}$ and their commonality specification. Fig. 5(i) shows the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules.

In $\mathfrak{MD}$ the atomic constraints $([\texttt{mult}(0,3)], \delta_1)$ and $([\texttt{mult}(1,4)], \delta_1)$ are annotated with $\texttt{<add>}^{[\texttt{mult}(m,n)]}$, which is visualised by a ++ and green colouring. In $\mathfrak{MD}''$ these atomic constraints are additionally annotated with $\texttt{<conflict>}^{[\texttt{mult}(m,n)]}$, which is visualised by a dotted box, according to rule l (see Table 3).

[1] The choice of the notation $\mathfrak{MD}''$ rather than $\mathfrak{MD}'$ will be clear in Section 4.4

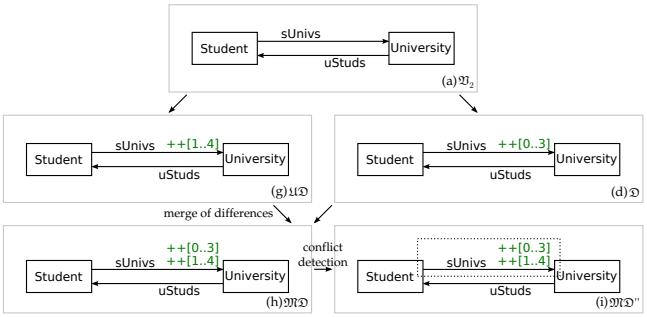


Figure 5: The merge of differences $\mathfrak{MD}$ and the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules

4.3 Conflict resolution

Depending on the structure and semantics of the modifications, some conflicts may be automatically resolved. Several *resolution strategies* [9] may be possible for a given conflict. These strategies are specified by *conflict resolution patterns*, which are based on transformation rules. A conflict resolution pattern consists of a transformation rule, where the left-hand side $\mathfrak{L}$ represents the conflict, the right-hand side $\mathfrak{R}$ is a specification where the resolution is applied, and $\mathfrak{K}$ is their interface.

In the merge of differences $\mathfrak{MD}$, we could annotate modifications from different developers with different labels so that a conflict resolution with priorities could take this information into account. However, in this paper, we abstract from these details and consider resolution patterns without priorities.

Resolving a conflict consists of applying a conflict resolution pattern by finding a match for the left-hand side $\mathfrak{L}$ in the merge of differences $\mathfrak{MD}''$, leading to a target merge of differences $\mathfrak{MD}'''$. Hence, in addition to conflict detection rules, the merge of differences $\mathfrak{MD}$ is processed by applying all applicable conflict resolution patterns.

To resolve conflicts of inconsistent multiplicity constraints, two conflict resolution patterns are defined. The first "liberal" pattern b_L is to remove the conflicting multiplicity constraints and add a constraint that is the union of the two. The second "conservative" pattern b_C is to remove the conflicting multiplicity constraints and add a constraint that is the intersection of the two. Table 3 shows these conflict resolution patterns.

In b_L, according to the semantic interpretation $[\![[\texttt{mult}(m,n)]]\!]^{\Sigma}$ of the signature Σ (see Table 1), the set of valid instances of the atomic constraint $([\texttt{mult}(min(m_1,m_2), max(n_1,n_2))], \delta_1) \in C^{\mathfrak{R}}$ is equal to the union of the set of valid instances of the atomic constraints $([\texttt{mult}(m_1,n_1)], \delta_1)$, $([\texttt{mult}(m_2,n_2)], \delta_2) \in C^{\mathfrak{L}}$. This is justified as follows:

$$([\texttt{mult}(m_1,n_1)], \delta) \wedge ([\texttt{mult}(m_2,n_2)], \delta) \equiv$$

$$\equiv \begin{cases} ([\texttt{mult}(m_2,n_2)], \delta) & \text{if} \quad m_1 \le m_2 \le n_2 \le n_1 \\ ([\texttt{mult}(m_1,n_1)], \delta) & \text{if} \quad m_2 \le m_1 \le n_1 \le n_2 \\ ([\texttt{mult}(m_2,n_1)], \delta) & \text{if} \quad m_1 \le m_2 \le n_1 \le n_2 \\ ([\texttt{mult}(m_1,n_2)], \delta) & \text{if} \quad m_2 \le m_1 \le n_2 \le n_1 \end{cases}$$

Table 3: The conflict detection rules and resolution patterns for inconsistent multiplicity constraints

Rule	$\mathfrak{L}$	$\mathfrak{K}$	$\mathfrak{R}$
g	X —(++[m..n], - -f)→ Y		X —(++[m..n], - -f)→ Y
l	X —(++[m_1..n_1] ++[m_2..n_2], f)→ Y		X —(++[m_1..n_1] ++[m_2..n_2], f)→ Y
b_L	X —(++[m_1..n_1] ++[m_2..n_2], f)→ Y	X —(f)→ Y	X —(++[min(m_1,m_2)..max(n_1,n_2)], f)→ Y
b_C	X —(++[m_1..n_1] ++[m_2..n_2], f)→ Y	X —(f)→ Y	X —(++[max(m_1,m_2)..min(n_1,n_2)], f)→ Y

Similarly, in b_C, the set of valid instances of the atomic constraint $([\texttt{mult}(max(m_1, m_2), min(n_1, n_2))], \delta_1) \in C^{\mathfrak{R}}$ is equal to the intersection of the set of valid instances of the atomic constraints $([\texttt{mult}(m_1, n_1)], \delta_1)$, $([\texttt{mult}(m_2, n_2)], \delta_2) \in C^{\mathfrak{L}}$. This is justified as follows:

$$([\texttt{mult}(m_1, n_1)], \delta) \vee ([\texttt{mult}(m_2, n_2)], \delta) \equiv$$

$$\equiv \begin{cases} ([\texttt{mult}(m_1, n_1)], \delta) & \text{if} \quad m_1 \leq m_2 \leq n_2 \leq n_1 \\ ([\texttt{mult}(m_2, n_2)], \delta) & \text{if} \quad m_2 \leq m_1 \leq n_1 \leq n_2 \\ ([\texttt{mult}(m_1, n_2)], \delta) & \text{if} \quad m_1 \leq m_2 \leq n_1 \leq n_2 \\ ([\texttt{mult}(m_2, n_1)], \delta) & \text{if} \quad m_2 \leq m_1 \leq n_2 \leq n_1 \end{cases}$$

Note that the conflict resolution patterns b_L and b_C can be applied only under the condition that the range of the multiplicity constraints overlap, i.e., if $n_1 \geq m_2$ or $m_1 \leq n_2$. This could be formulated as a positive application condition [21].

The following example illustrates the application of conflict resolution patterns.

Example 4.3 (Conflict resolution). Building on Example 2.1, Fig. 6(i) shows the merge of differences $\mathfrak{MD}''$, while Fig.s 6(j) and 6(k) show the merge of differences $\mathfrak{MD}'''$ after the application of the liberal and conservative conflict resolution patterns, respectively.

In $\mathfrak{MD}''$ the atomic constraints $([\texttt{mult}(0,3)], \delta_1)$ and $([\texttt{mult}(1,4)], \delta_1)$ are annotated with $\texttt{<add>}^{[\texttt{mult}(m,n)]}$ and $\texttt{<conflict>}^{[\texttt{mult}(m,n)]}$. In $\mathfrak{MD}'''$ these atomic constraints are replaced with a new atomic constraint $([\texttt{mult}(0,4)], \delta_1)$, according to pattern b_L, or $([\texttt{mult}(1,3)], \delta_1)$, according to pattern b_C (see Table 3).

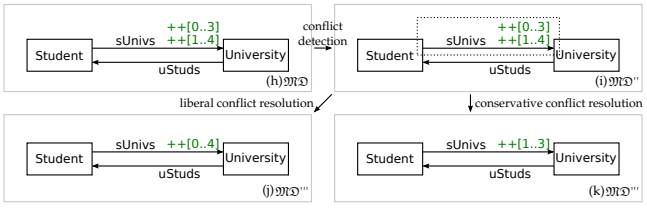


Figure 6: The merge of differences $\mathfrak{MD}''$ and the merge of differences $\mathfrak{MD}'''$ after the application of the conflict resolution patterns

4.4 Normalisation, conflict detection and conflict resolution

In general, the merge of differences $\mathfrak{MD}$ is a valid specification by construction, but it may not be in *normal form*; i.e., single atomic constraints of the specification may not express syntactically the constraints that the conjunction of all the atomic constraints implies semantically. Performing conflict detection on a merge of differences that is not in normal form may lead to a specification containing false negatives [29]; i.e., containing actual conflicts that are not annotated with `conflict`. Moreover, performing conflict resolution on the merge of differences that is not in normal form may lead to a specification that is also not in normal form.

Intuitively, the normal form is a specification $\mathfrak{N}$ where there exist specification morphisms $\phi : \mathfrak{N} \rightarrow \{\mathfrak{MD}''\}$ to all other possible ways of applying resolution patterns to $\mathfrak{MD}$. This specification may not always be unique. Therefore, we consider the scenario where we obtain a reasonable normal form, which contains the minimal set of constraints that give the same semantics. The interested reader may refer to [33] for a formal definition of the normal form.

The following example illustrates an alternative scenario of concurrent development in MDSE, which leads to a merge of differences $\mathfrak{MD}$ that is not in normal form.

Example 4.4 (Alternative custom conflict detection). Let us consider a variant of the scenario in Example 2.1. Fig. 7 shows the different versions of the specification being developed.

In addition to the atomic constraint $([\texttt{mult}(1,4)], \delta_1)$ on the arrow sUnivs, Alice adds the atomic constraints $([\texttt{surjective}], \delta_3)$ on the arrows uStuds and $([\texttt{inverse}], \delta_2)$ on the arrows sUnivs and uStuds.

Fig. 7(h) shows the merge of differences $\mathfrak{MD}$, Fig. 7(j) shows the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules, while Fig.s 7(k) and 7(l) show the merge of differences $\mathfrak{MD}'''$ after the application of the liberal and conservative conflict resolution patterns, respectively.

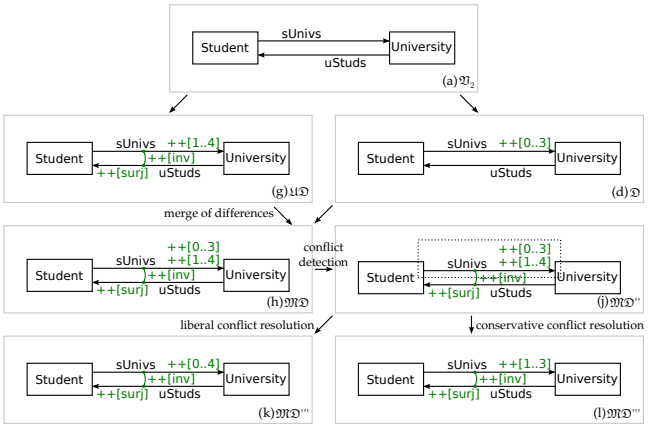


Figure 7: The merge of differences $\mathfrak{M}\mathfrak{D}$, the merge of differences $\mathfrak{M}\mathfrak{D}''$ after the application of conflict detection rules and the merge of differences $\mathfrak{M}\mathfrak{D}'''$ after the application of the conflict resolution patterns

The application of the conflict resolution pattern b_L for inconsistent multiplicity constraints (see Table 3) leads to a merge of differences $\mathfrak{M}\mathfrak{D}'''$ that is not in normal form. In fact, the single atomic constraint ([mult(0, 4)], δ_1) on the arrow sUnivs expresses syntactically that the function *sUnivs* has cardinality between zero and four (see Fig. 7(k)), but the conjunction of all the atomic constraints ([mult(0, 4)], δ_1), ([inverse], δ_2) and ([surjective], δ_3) implies semantically that the function *sUnivs* has cardinality between one and four. This is justified as follows:

$$\begin{aligned} uStuds \text{ surjective} &\Rightarrow \forall s \in Student\ \exists u \in University \\ &\quad \text{where } x \in uStuds(y) \\ sUnivs, uStuds \text{ inverse} &\Rightarrow \forall s \in Student\ \exists u \in University \\ &\quad \text{where } y \in sUnivs(x) \\ sUnivs \text{ total} &\Rightarrow \forall s \in Student : |sUnivs(x)| \geq 1 \end{aligned}$$

In this paper, the properties of conjunctively connected sets of atomic constraints are described by means of *specification entailments*. A specification entailment has the structure $Left \vdash Right$, where both premise ($Left$) and conclusion ($Right$) are specifications with the same underlying graph. From a specification entailment, one may induce a transformation rule that can be applied to existing specifications. The interested reader may consult [33, 36] for a more detailed discussion of specification entailments.

The following example illustrates the usage of specification entailments.

Example 4.5 (Specification entailment). Fig. 8 shows a specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$ that expresses the relation between multiplicity and surjectivity constraints.

Table 4 shows the transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ induced by the corresponding specification entailment e. Fig. 9(b) shows the specification $\mathfrak{S}' = (S', C^{\mathfrak{S}'} : \Sigma)$ after the application of the transformation rule t.

Performing conflict detection on a merge of differences $\mathfrak{M}\mathfrak{D}$ that is not in normal form may lead to a merge of differences $\mathfrak{M}\mathfrak{D}''$ containing false negatives or false positives [29]. To ensure that

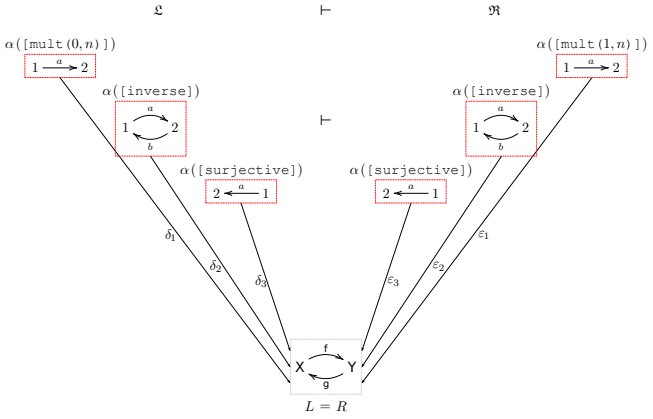


Figure 8: A specification entailment $e = \mathfrak{L} \vdash \mathfrak{R}$

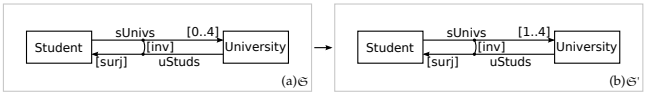


Figure 9: The application of the transformation rule t

conflict detection and conflict resolution behave as expected, they are performed on a merge of differences $\mathfrak{M}\mathfrak{D}$ in normal form.

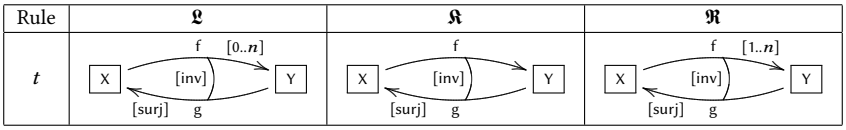
In this paper, normalisation consists of a sequence of applications of transformation rules induced by specification entailments. More precisely, given a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ and a set of transformation rules induced by specification entailments, a normalisation consists of a specification transformation $\mathfrak{S} \overset{*}{\Longrightarrow} \mathfrak{S}'$, leading to a normal form $\mathfrak{S}'$. Note that the normalisation is assumed to be terminating and confluent; i.e., each specification can be transformed to a unique normal form by specification transformation. The identification of the conditions under which a set of specification entailments guarantees termination and confluence of the normalisation is outside the scope of this work (see Section 6).

The following example illustrates the application of normalisation.

Example 4.6 (Normalisation, conflict detection and conflict resolution). Building on Example 4.4, Fig. 10(i) shows the merge of differences $\mathfrak{M}\mathfrak{D}'$ after the normalisation, Fig. 10(j) shows the merge of differences $\mathfrak{M}\mathfrak{D}''$ after the application of conflict detection rules, while Fig.s 10(k) and 10(l) show the merge of differences $\mathfrak{M}\mathfrak{D}'''$ after the application of the liberal and conservative conflict resolution patterns, respectively.

The normalisation replaces the atomic constraint ([mult(0, 3)], δ_1) in $\mathfrak{M}\mathfrak{D}$ with ([mult(1, 3)], δ_1) in $\mathfrak{M}\mathfrak{D}'$. As a consequence, the application of the conflict resolution pattern b_L (see Table 3) leads to a merge of differences $\mathfrak{M}\mathfrak{D}'''$ that is in normal form.

Table 4: The transformation rule $t = \mathfrak{L} \leftarrow \mathfrak{K} \hookrightarrow \mathfrak{R}$ induced by the corresponding specification entailment e



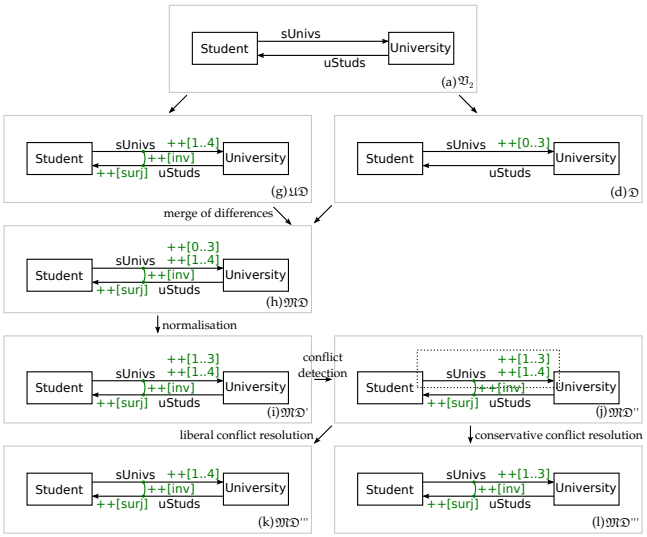


Figure 10: The merge of differences $\mathfrak{MD}$, the merge of differences $\mathfrak{MD}'$ after the normalisation, the merge of differences $\mathfrak{MD}''$ after the application of conflict detection rules and the merge of differences $\mathfrak{MD}'''$ after the application of the conflict resolution patterns

5 RELATED WORK

Model versioning has been greatly discussed in the literature. The interested reader may consult [4, 7] for a survey and an introduction.

A strand of research within model versioning focuses on the problem of model merging. Different approaches can be found in the literature:

- In [13], the authors propose a search-based automated model merge approach where rule-based design space exploration is used to search the space of solution candidates that represent conflict-free merged models. Similar to our approach, this work takes into consideration certain structural and numeric constraints.
- The work in [12] uses a repair generator to address inconsistencies that are produced while merging architectural models. The work in [11] examines the same issues more thoroughly and uses search-based techniques.
- In [31], the authors propose to use an artificial intelligence technique of automated planning for resolving inconsistencies—be it due to model merging or due to model editions—through the generation of one or more resolution plans. They also implement the tool Badger in Prolog, which is a regression planner generating such plans.
- The work in [43] presents a formal approach to the three-way merging of Ecore [40] models based on set theory and predicate logic. It is based on formally defined merge rules that can handle additions, deletions and renames of model elements as well as moves of contained model elements. Furthermore, it detects and resolves conflicting modifications of the same element and of different interdependent elements. Finally, the approach guarantees that the resulting merged model is well-formed.
- The work in [41] proposes a formal approach to the merging of typed attributed graphs based on graph transformations and category theory. In this approach, two kinds of conflicts are defined based on the notion of graph modifications: operation-based and state-based conflicts. On the one hand, operation-based conflicts are detected by first extracting minimal rules from modifications and thereafter, if possible, selecting pre-defined operation rules. Conflict detection is then based on the parallel dependence of graph transformations and extraction of critical pairs. On the other hand, state-based conflicts are detected by checking the merged graphs against graph constraints.
- In [10], the authors propose a technique for obtaining automatically generated repair plans for a given inconsistent model. Repair plans are sequences of concrete modifications to be performed on a given model that fix existing inconsistencies without introducing new ones. The technique is based on Praxis, which is a model inconsistency detection approach. In Praxis, the model is represented as the sequence of actions executed by the user in order to build it.
- The work in [9] introduces a domain-specific modelling language for the definition of weaving models that represent patterns of conflicting modifications. Resolution criteria for these patterns can be specified through OCL expressions.

With the exception of [13], the approaches mentioned above do not take constraints on model elements into account. However, the approaches in [10, 41, 43] include checking the well-formedness of the result of merging. In future work, we may also explore this important dimension in our approach to model versioning.

Research has also lead to a number of prototype tools that support model versioning:

- EMFStore [27] provides a dedicated framework for version control of EMF models. It is an operation- and graph-based approach that supports the detection of composite modifications.
- Adaptable Model Versioning (AMOR) [8] is a versioning system that can deal with arbitrary modelling languages based on Ecore. AMOR is built around Subversion to provide a centralised approach to optimistic versioning, but reuses an

extended version of EMF Compare [22] for difference calculation. AMOR provides conflict detection features that may be enhanced with user-defined operations. Moreover, it provides collaborative conflict resolution features, which allow the implementation of conflict resolution policies. If the resolution is performed manually, it is analysed to derive resolution recommendations for similar situations that occur in future scenarios. In contrast to our approach, this tool does not provide a formal treatment of conflict detection and resolution.

- Semantically enhanced Model Version Control System (SMoVer) [3] facilitates the specification of modelling language semantics, which is needed for accurate conflict detection. The authors exemplify how semantics can improve the accuracy of conflict detection and how these conflicts can be presented to modellers. On the one hand, certain models that are in conflict syntactically may be merged without conflicts based on their semantics. On the other hand, certain models that are not in conflict syntactically may not be merged without conflicts based on their semantics. Similar to our approach, this tool enables the definition of rules for conflict detection and resolution based on both syntax and semantics.
- Epsilon Comparison Language (ECL) and Epsilon Merging Language (EML) [30] provide a rule-based language for comparing and merging homogeneous or heterogeneous models, respectively. Rules specified in ECL and EML could be employed for conflict detection and resolution.

An Eclipse-based modelling environment for DPF is described in [28], while a web-based modelling environment for DPF is presented in [32]. In future work, we may implement prototype support for our approach to model versioning and perform case studies to compare the existing tools with our tool.

An initial attempt in this direction—based on Resource Description Format (RDF)—is described in [39]. Moreover, the DPF modelling environment utilises Alloy [26] for consistency checking [42]. Alloy could be used to check the satisfiability of the merge of differences, which in turn could be used in the process of normalisation and conflict detection.

6 CONCLUSION AND FUTURE WORK

In this paper, we described a formal approach to model versioning based on DPF. First, we defined the representation of differences—i.e., the information added, deleted and renamed—as a set of annotations that are specified by means of a label signature. Second, we introduced a synchronisation procedure that includes normalisation, conflict detection and conflict resolution. Transformation rules are used to represent conflicts and—where applicable—their resolution patterns. The conflict detection and resolution are then formalised as the application of these transformation rules. Moreover, specification entailments are adopted to describe properties of conjunctively connected sets of atomic constraints. The normalisation of a specification is then formalised as the embedding of these specification entailments to obtain the normal form of a specification.

Note that the proposed approach handles constraints in all the steps of the synchronisation, including normalisation, conflict detection and conflict resolution. To the best of our knowledge, this work constitutes the first attempt to formalise constraint-awareness in model versioning.

Specification transformations constitute the basis for normalisation, conflict detection and conflict resolution. In future work, we will analyse termination and confluence in DPF. This will facilitate the identification of the conditions under which a set of conflict resolution patterns guarantees that no new conflicts are introduced.

REFERENCES

[1] Kerstin Altmanninger. 2009. Issues and challenges in model versioning. In *iiWAS 2009: 11th International Conference on Information Integration and Web-based Applications and Services*, Gabriele Kotsis, David Taniar, Eric Pardede, and Ismail Khalil (Eds.). ACM, 8. https://doi.org/10.1145/1806338.1806344

[2] Kerstin Altmanninger, Petra Brosch, Philip Langer, Martina Seidl, Konrad Wiel, and Manuel Wimmer. 2009. Why Model Versioning Research is Needed!? An Experience Report. In *Models and Evolution: Joint MoDSE-MCCM 2010 Workshop on Model-Driven Software Evolution and Model Co-Evolution and Consistency Management*. 1–12.

[3] Kerstin Altmanninger, Wieland Schwinger, and Gabriele Kotsis. 2010. Semantics for Accurate Conflict Detection in SMoVer: Specification, Detection and Presentation by Example. *International Journal of Enterprise Information Systems* 6, 1 (2010), 68–84. https://doi.org/10.4018/jeis.2010120206

[4] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. 2009. A Survey on Model Versioning Approaches. *International Journal of Web Information Systems* 5, 3 (2009), 271–304. https://doi.org/10.1108/17440080910983556

[5] Apache Subversion. Accessed 2018-08-20. https://subversion.apache.org/

[6] Michael Barr and Charles Wells. 1995. *Category Theory for Computing Science (2^{nd} Edition)*. Prentice Hall.

[7] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. 2012. An Introduction to Model Versioning. In *SFM 2012: Formal Methods for Model-Driven Engineering—12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (Lecture Notes in Computer Science)*, Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio (Eds.), Vol. 7320. Springer, 336–398. https://doi.org/10.1007/978-3-642-30982-3_10

[8] Petra Brosch, Gerti Kappel, Martina Seidl, Konrad Wieland, Manuel Wimmer, Horst Kargl, and Philip Langer. 2010. Adaptable Model Versioning in Action. In *Modellierung 2010 (Lecture Notes in Informatics)*, Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr (Eds.), Vol. 161. GI, 221–236.

[9] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. 2008. Managing Model Conflicts in Distributed Development. In *MoDELS 2008: 11th International Conference on Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science)*, Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter (Eds.), Vol. 5301. Springer, 311–325. https://doi.org/10.1007/978-3-540-87875-9_23

[10] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. 2010. Towards Automated Inconsistency Handling in Design Models. In *CAiSE 2010: 22nd International Conference on Advanced Information Systems Engineering (Lecture Notes in Computer Science)*, Barbara Pernici (Ed.), Vol. 6051. Springer, 348–362. https://doi.org/10.1007/978-3-642-13094-6_28

[11] Hoa Khanh Dam, Alexander Egyed, Michael Winikoff, Alexander Reder, and Roberto E. Lopez-Herrejon. 2016. Consistent merging of model versions. *Journal of Systems and Software* 112 (2016), 137–155. https://doi.org/10.1016/j.jss.2015.06.044

[12] Hoa Khanh Dam, Alexander Reder, and Alexander Egyed. 2014. Inconsistency Resolution in Merging Versions of Architectural Models. In *WICSA 2014: 2014 IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society, 153–162. https://doi.org/10.1109/WICSA.2014.31

[13] Csaba Debreceni, István Ráth, Dániel Varró, Xabier De Carlos, Xabier Mendialdua, and Salvador Trujillo. 2016. Automated Model Merge by Design Space Exploration. In *FASE 2016: 19th International Conference (Lecture Notes in Computer Science)*, Perdita Stevens and Andrzej Wasowski (Eds.), Vol. 9633. Springer, 104–121. https://doi.org/10.1007/978-3-662-49665-7_7

[14] Zinovy Diskin. 2002. Visualization vs. Specification in Diagrammatic Notations: A Case Study with the UML. In *Diagrams 2002: 2nd International Conference on Diagrammatic Representation and Inference (Lecture Notes in Computer Science)*, Mary Hegarty, Bertrand Meyer, and N. Hari Narayanan (Eds.), Vol. 2317. Springer, 112–115. https://doi.org/10.1007/3-540-46037-3_15

[15] Zinovy Diskin. 2003. *Practical foundations of business system specifications.* Springer, Chapter Mathematics of UML: Making the Odysseys of UML less dramatic, 145–178.

[16] Zinovy Diskin. 2005. *Encyclopedia of Database Technologies and Applications.* Information Science Reference, Chapter Mathematics of Generic Specifications for Model Management I and II, 351–366.

[17] Zinovy Diskin and Boris Kadish. 2003. Variable set semantics for keyed generalized sketches: formal semantics for object identity and abstract syntax for conceptual modeling. *Data & Knowledge Engineering* 47, 1 (2003), 1–59. https://doi.org/10.1016/S0169-023X(03)00047-8

[18] Zinovy Diskin and Boris Kadish. 2005. *Encyclopedia of Database Technologies and Applications.* Information Science Reference, Chapter Generic Model Management, 258–265.

[19] Zinovy Diskin, Boris Kadish, Frank Piessens, and Michael Johnson. 2000. Universal Arrow Foundations for Visual Modeling. In *Diagrams 2000: 1st International Conference on Diagrammatic Representation and Inference (Lecture Notes in Computer Science)*, Michael Anderson, Peter Cheng, and Volker Haarslev (Eds.), Vol. 1889. Springer, 345–360. https://doi.org/10.1007/3-540-44590-0_30

[20] Zinovy Diskin and Uwe Wolter. 2008. A Diagrammatic Logic for Object-Oriented Visual Modeling. In *ACCAT 2007: 2nd Workshop on Applied and Computational Category Theory (Electronic Notes in Theoretical Computer Science)*, Vol. 203/6. Elsevier, 19–41. https://doi.org/10.1016/j.entcs.2008.10.041

[21] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation.* Springer. https://doi.org/10.1007/3-540-31188-2

[22] EMF Compare. Accessed 2018-08-20. https://www.eclipse.org/emf/compare/

[23] José Luiz Fiadeiro. 2004. *Categories for Software Engineering.* Springer.

[24] Git. Accessed 2018-08-20. https://git-scm.com

[25] James W. Hunt and M. D. McIlroy. 1976. *An Algorithm for Differential File Comparison.* Technical Report 41. Bell Laboratories.

[26] Daniel Jackson. 2012. *Software abstractions: logic, language, and analysis.* MIT Press.

[27] Maximilian Koegel and Jonas Helming. 2010. EMFStore: a model repository for EMF models. In *ICSE 2010: 32nd ACM/IEEE International Conference on Software Engineering*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 307–308. https://doi.org/10.1145/1810295.1810364

[28] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Wendy MacCaull, and Adrian Rutle. 2012. DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment. In *Computer and Information Science*, Roger Lee (Ed.). Studies in Computer Intelligence, Vol. 429. Springer, 37–52. https://doi.org/10.1007/978-3-642-30454-5_3

[29] Tom Mens. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462. https://doi.org/10.1109/TSE.2002.1000449

[30] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nikolaos Drivalos, and Fiona A. C. Polack. 2009. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *ICECCS 2009: 14th IEEE International Conference on Engineering of Complex Computer Systems.* IEEE Computer Society, 162–171. https://doi.org/10.1109/ICECCS.2009.14

[31] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. 2015. Resolving model inconsistencies using automated regression planning. *Software and System Modeling* 14, 1 (2015), 461–481. https://doi.org/10.1007/s10270-013-0317-9

[32] Fazle Rabbi, Yngve Lamo, Ingrid Chieh Yu, and Lars Michael Kristensen. 2016. WebDPF: A Web-based Metamodelling and Model Transformation Environment. In *MODELSWARD 2016: 4th International Conference on Model-Driven Engineering and Software Development*, Slimane Hammoudi, Luís Ferreira Pires, Bran Selic, and Philippe Desfray (Eds.). SciTePress, 87–98. https://doi.org/10.5220/0005686900870098

[33] Alessandro Rossini. 2011. *Diagram Predicate Framework meets Model Versioning and Deep Metamodelling.* Ph.D. Dissertation. Department of Informatics, University of Bergen, Norway.

[34] Alessandro Rossini, Juan de Lara, Esther Guerra, Adrian Rutle, and Uwe Wolter. 2014. A formalisation of deep metamodelling. *Formal Aspects of Computing* 26, 6 (2014), 1115–1152. https://doi.org/10.1007/s00165-014-0307-x

[35] Alessandro Rossini, Adrian Rutle, Yngve Lamo, and Uwe Wolter. 2010. A formalisation of the copy-modify-merge approach to version control in MDE. *Journal of Logic and Algebraic Programming* 79, 7 (2010), 636–658. https://doi.org/10.1016/j.jlap.2009.10.003

[36] Adrian Rutle. 2010. *Diagram Predicate Framework: A Formal Approach to MDE.* Ph.D. Dissertation. Department of Informatics, University of Bergen, Norway.

[37] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. 2009. A Category-Theoretical Approach to the Formalisation of Version Control in MDE. In *FASE 2009: 12th International Conference on Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Marsha Chechik and Martin Wirsing (Eds.), Vol. 5503. Springer, 64–78. https://doi.org/10.1007/978-3-642-00593-0_5

[38] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. 2012. A formal approach to the specification and transformation of constraints in MDE. *Journal of Logic and Algebraic Programming* 81, 4 (2012), 422–457. https://doi.org/10.1016/j.jlap.2012.03.006

[39] Hans Georg Schaathun and Adrian Rutle. 2018, to appear. Model-Driven Software Engineering in the Resource Description Framework: a way to version control. In *NIK 2018: 31st Norsk Informatikkonferanse.* BIBSYS Open Journal System.

[40] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework 2.0 (2nd Edition).* Addison-Wesley Professional.

[41] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. 2010. Conflict Detection for Model Versioning Based on Graph Modifications. In *ICGT 2010: 5th International Conference on Graph Transformations (Lecture Notes in Computer Science)*, Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr (Eds.), Vol. 6372. Springer, 171–186. https://doi.org/10.1007/978-3-642-15928-2_12

[42] Xiaoliang Wang, Adrian Rutle, and Yngve Lamo. 2015. Towards User-Friendly and Efficient Analysis with Alloy. In *MoDeVVa@MoDELS 2015: 12th Workshop on Model-Driven Engineering, Verification and Validation (CEUR Workshop Proceedings)*, Michalis Famelis, Daniel Ratiu, Martina Seidl, and Gehan M. K. Selim (Eds.), Vol. 1514. CEUR-WS.org, 28–37. http://ceur-ws.org/Vol-1514

[43] Bernhard Westfechtel. 2010. A Formal Approach to Three-Way Merging of EMF Models. In *IWMCP 2010: 1st International Workshop on Model Comparison in Practice.* ACM, 31–41. https://doi.org/10.1145/1826147.1826155