Towards Development with Multi-version Models: Detecting Merge Conflicts and Checking Well-Formedness

Matthias Barkowsky(✉) and Holger Giese

Hasso-Plattner Institute at the University of Potsdam,
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{matthias.barkowsky,holger.giese}@hpi.de

Abstract. Developing complex software requires that multiple views and versions of the software can be developed in parallel and merged as supported by views and managed by version control systems. In this context, this paper considers permanent monitoring of merging and related consistency problems at the level of models and abstract syntax. The presented approach introduces multi-version models based on typed graphs that permit to store changes and multiple versions in one graph in a compact form and allow (1) to study well-formedness for all versions without the need to extract each version individually, (2) to report all possible merge conflicts without the need to merge all pairs of versions, and (3) to report all violations of well-formedness conditions that will result for merges of any two versions independent of any merge decisions without the need to merge all pairs of versions. Thereby, the approach aims to permit early and frequent conflict detection while developing in parallel. The paper defines the related concepts and algorithms operating on multi-version models, proves their correctness w.r.t. the usually employed three-way-merge, and reports on preliminary experiments concerning the scalability.

1 Introduction

Developing complex software nowadays requires that multiple views and versions of the software can be developed in parallel and merged as supported by views and managed by version control systems [12]. For complex software, living with inconsistencies at least temporarily is inevitable, as enforcing consistency may lead to loss of important information [11] and is hence neither always possible nor desirable. However, working with multiple versions in parallel and changing each version on its own for longer periods of time can introduce substantial conflicts that are difficult and expensive to resolve. Therefore, it is necessary to manage consistency when combining views and versions using merge approaches [12,20].

This paper considers permanent monitoring of merging and related consistency problems at the level of models and abstract syntax. This aims to permit

This work was developed mainly in the course of the project modular and incremental Global Model Management (project number 336677879) funded by the DFG.

N. Behr and D. Strüber (Eds.): ICGT 2022, LNCS 13349, pp. 118–136, 2022.
https://doi.org/10.1007/978-3-031-09843-7_7

early and frequent conflict detection while developing in parallel, as suggested in approaches to detect conflicts early and to enable collaboration to manage conflicts and their risks [4].

The presented approach therefore introduces multi-version models based on typed graphs, which permit to store changes and multiple versions in one graph in a compact form and allow to study the different versions and their merge combinations. The following capabilities are considered: (1) Study well-formedness for all versions at once without the need to extract and explicitly consider each version individually. (2) Report all possible merge conflicts that may result for merges of any two versions without the need to extract and explicitly merge all pairs of versions. (3) Report all violations of well-formedness conditions that will result for merges of any two versions independent of any merge decisions without the need to extract and explicitly merge all pairs of versions.

The approach thus promises to support early conflict detection and collaboration for managing conflicts and their risks, while not having to decide how to later merge conflicting versions. The technique also aims for a better scalability in case there are many versions that are considered in parallel.

Furthermore, the developed multi-version models permit to study the phenomena of versions, merging, and well-formedness conditions in the unifying framework of typed graphs. This enables us to (a) formulate algorithms that can obtain several analysis results without the need to consider a specific version, merge of a pair of versions, or strategy for conflict resolution and (b) prove that the algorithms compute the same results as if we would explicitly consider all specific versions, merges of pairs of versions, or strategies for conflict resolution.

The paper defines the related concepts and algorithms operating on multi-version models, proves their correctness w.r.t. the usually employed three-way-merge, and reports on first experiments concerning the scalability. In Sect. 2, we summarize the preliminaries of the presented approach, including basic definitions for typed graphs, well-formedness conditions, and graph modifications. Then, as a baseline, single-version models in the form of typed graphs with well-formedness conditions are defined in Sect. 3, before multi-version models are introduced in Sect. 4. Determining all merge conflicts and checking well-formedness for all merge results based on multi-version models is then considered in Sect. 5. Results of first experiments for our prototypical implementation of the algorithms are presented in Sect. 6. A summary of related work is given in Sect. 7. Finally, the conclusions of the paper and an outlook of planned future work are presented in Sect. 8.

2 Preliminaries

We briefly reiterate the basic concepts of graphs, graph modifications, and well-formedness conditions used in the remainder of the paper.

A graph $G = (V^G, E^G, s^G, t^G)$ consists of a set of nodes V^G, a set of edges E^G and two functions $s^G : E^G \to V^G$ and $t^G : E^G \to V^G$ assigning each edge its source and target, respectively. We assume that graph elements have identities and source and target of an edge are invariant if an edge is part of multiple graphs, that is, for two graphs G and H and an edge $e \in E^G \cap E^H$, it holds that $s^G(e) = s^H(e)$ and $t^G(e) = t^H(e)$. This also implies that, in the context of this paper, $(V^G = V^H \wedge E^G = E^H) \rightarrow (G = H)$.

A graph morphism $m : G \rightarrow H$ is given by a pair of functions $m^V : V^G \rightarrow V^H$ and $m^E : E^G \rightarrow E^H$ that map elements from G to elements from H such that $s^H \circ m^E = m^V \circ s^G$ and $t^H \circ m^E = m^V \circ t^G$ [9].

A graph G can be typed over a type graph TG via a typing morphism $type : G \rightarrow TG$, forming the typed graph $G^T = (G, type^G)$. A typed graph morphism between two typed graphs $G^T = (G, type^G)$ and $H^T = (H, type^H)$ with the same type graph then denotes a graph morphism $m^T : G \rightarrow H$ such that $type^G = type^H \circ m^T$. A (typed) graph morphism m is a monomorphism iff its functions m^V and m^E are injective.

Figure 1 shows an example typed graph M_1 and associated type graph TM from the software development domain. M_1 represents an abstract syntax graph for a program written in an object-oriented language that contains four classes represented by nodes. The type graph also allows representing superclass relationships with edges.

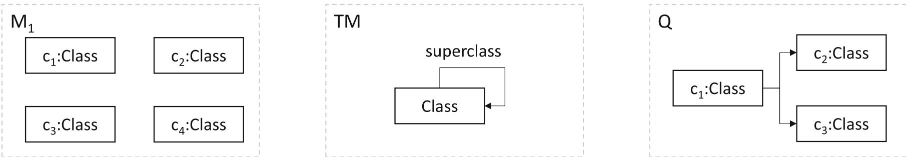

Fig. 1. Example graph, type graph, and violation pattern

The structure of a typed graph G can be restricted by a well-formedness condition ϕ, which in the context of this paper is characterized by a typed graph Q typed over the same type graph. G then satisfies the condition ϕ, denoted $G \models \phi$, iff there exists no monomorphism $m : Q \rightarrow G$. We also call such monomorphisms *matches* and Q the *violation pattern* of ϕ.

Figure 1 shows a violation pattern Q for an example well-formedness constraint that forbids a class having two outgoing superclass relationships.

A graph modification as defined by Taentzer et al. [26] formalizes the difference between two graphs G and H and is characterized by an intermediate graph K and a span of monomorphisms $(G \leftarrow K \rightarrow H)$. In this paper, we assume that the two morphisms are always subgraph inclusions. K then characterizes the subgraph that is preserved through the modification, whereas elements in G that are not in K are deleted and elements in H but not in K are created.

Figure 2 shows an example graph modification from the graph M_1 from Fig. 1 to a new graph M_2, where a superclass edge from class c_1 to class c_3 is created and the class c_4 is deleted. The morphisms are implied by node labels.

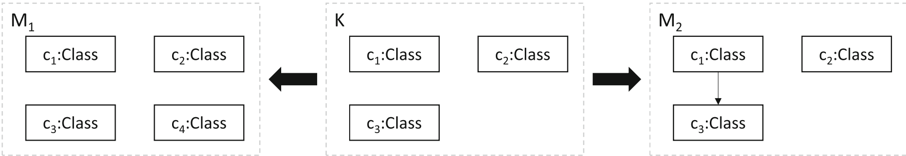

Fig. 2. Example graph modification

Graphs and graph modifications correspond to versions and differences in conventional, line-based version control systems like Git [16], where versions of a development artifact and intermediate differences form a directed acyclic graph.

3 Single-Version Models

In this paper, we consider models in the form of typed graphs that are required to adhere to a set of well-formedness conditions. Effectively, the combination of type graph and well-formedness conditions then acts as a metamodel with potential further constraints. Note that attributes, as usually employed in real-world models, can in this context be modeled as dedicated nodes [17].

For Φ the set of well-formedness conditions, a model M_i is *well-formed* iff $\forall \phi \in \Phi : M_i \models \phi$. We assume $pcheck(M_i, \phi)$ to report all violations to property ϕ with violation pattern Q for model M_i in the form of matches for Q, essentially realizing $\models$ as $pcheck(M_i, \phi) = \emptyset \iff M_i \models \phi$. If violations exist, the model M_i is also called *ill-formed*.

For the notion of models as typed graphs, model modifications correspond to graph modifications as presented in Sect. 2. We say a model modification $(M_i \leftarrow K \rightarrow M_j)$ with subgraph inclusions is *maximally preserving* iff it does not delete and recreate identical elements. Formally, $K = (V^{M_i} \cap V^{M_j}, E^{M_i} \cap E^{M_j}, s^K, t^K)$, where s^K and t^K are uniquely defined assuming invariant edge sources and targets. Consequently, for two models M_i and M_j, the maximally preserving model modification $(M_i \leftarrow K \rightarrow M_j)$ is uniquely defined.

For a set of model modifications $\Delta^{M_{\{1,\dots,n\}}}$ between models $M_{\{1,\dots,n\}} = \{M_1, \dots, M_n\}$, with $\forall (G \leftarrow K \rightarrow H) \in \Delta^{M_{\{1,\dots,n\}}} : G \in M_{\{1,\dots,n\}} \wedge H \in M_{\{1,\dots,n\}}$, we can define the set of predecessors $pre(i) \subset M_{\{1,\dots,n\}}$ of a version M_i as the set of versions M_j such that there exists a sequence of model modifications $(M_{x_1} \leftarrow K_{x_1} \rightarrow M_{x_2})$, $(M_{x_2} \leftarrow K_{x_2} \rightarrow M_{x_3})$, $\dots$, $(M_{x_{n-1}} \leftarrow K_{x_{n-1}} \rightarrow M_{x_n})$ where $x_1 = j$, $x_n = i$, and $(M_{x_k} \leftarrow K_{x_k} \rightarrow M_{x_{k+1}}) \in \Delta^{M_{\{1,\dots,n\}}}$ for $1 \leq k < n$.

$\Delta^{M_{\{1,\dots,n\}}}$ describes a *correct version history* if all morphisms in the individual model modifications are subgraph inclusions, all model modifications are maximally preserving, the *pre* relation is acyclic and there exists a model M_α such that $M_\alpha \in pre(i)$ for all models $M_i \neq M_\alpha$. Effectively, a correct version history describes a directed acyclic graph of model versions $M_{\{1,\dots,n\}}$ that are derived from an original model M_α via the model modifications in $\Delta^{M_{\{1,\dots,n\}}}$, and therefore closely corresponds to the versioning of some development artifact in a conventional version control system.

Taentzer et al. [26] define a merge operation for model modifications $m_1 = (M_c \leftarrow K_i \rightarrow M_i)$ and $m_2 = (M_c \leftarrow K_j \rightarrow M_j)$ with common source M_c, which unifies m_1 and m_2 into a merged model modification $m_m = merge(m_1, m_2) = (M_c \leftarrow K_m \rightarrow M_m)$. We denote the merged model by $M_m = merge_G(m_1, m_2)$. This merge operation is similar to a three-way-merge in conventional version control systems [20], since m_m in the default case (i) preserves an element $x \in M_c$ iff it is preserved by both m_1 and m_2 (ii) deletes an element $x \in M_c$ iff it is deleted by m_1 or m_2 (iii) creates an element $x \in M_m$ iff it is created by m_1 or m_2.

However, according to [26], model modifications can be in conflict in two cases: (i) insert-delete conflict and (ii) delete-delete conflict. Taentzer et al. state that only (i), where one modification creates an edge connected to a node deleted by the other modification, is an actual conflict, which has to be resolved to create a correct merge result. In this case, the merge result may deviate from the default case. Such conflicts will be reported by $mcheck((M_c \leftarrow K_i \rightarrow M_i), (M_c \leftarrow K_j \rightarrow M_j))$ in the form (e, v), where e is an edge created by one of the modifications and v is a node deleted by the other modification.

For a correct version history $\Delta^{M_{\{1,\ldots,n\}}}$, we say that two sequences of model modifications $M_c \Rightarrow^* M_i$ and $M_c \Rightarrow^* M_j$ are in conflict iff their corresponding maximally preserving model modifications $(M_c \leftarrow K_{c,i} \rightarrow M_i)$ and $(M_c \leftarrow K_{c,j} \rightarrow M_j)$ are in conflict. In this case, we also say that M_i and M_j are in conflict for the common predecessor M_c.

Insert-delete conflicts can be resolved by equipping the *merge* operation with a manual or automatic strategy for conflict resolution. We consider such a strategy valid if it decides for each conflict whether to either revert the edge creation or the node deletion and always produces a proper merged graph. The approach in [26] effectively proposes an automatic strategy that favors insertion over deletion in order to preserve as many model elements as possible. Therefore, it reverts any deletions of nodes that would lead to insert-delete conflicts.

In contrast, a strategy for conflict resolution may favor deletion over insertion by reverting any creations of edges that would lead to insert-delete conflicts. Specifically, for model modifications $m_1 = (M_c \leftarrow K_i \rightarrow M_i)$ and $m_2 = (M_c \leftarrow K_j \rightarrow M_j)$, the model modification $m_{min} = merge^{min}(m_1, m_2)$, with $merge^{min}$ a merge operation equipped with this strategy, only creates an edge created by m_1 or m_2 if neither its source nor target is deleted by the other modification.

If all well-formedness conditions are specified by simple violation patterns, m_{min} also yields a model where all well-formedness violations are also present in the merge result for any other conflict resolution strategy:

Theorem 1. *For two model modifications $m_1 = (M_c \leftarrow K_i \rightarrow M_i)$ and $m_2 = (M_c \leftarrow K_j \rightarrow M_j)$ and a well-formedness constraint ϕ with violation pattern Q, it holds that*

$$pcheck(merge_G^{min}(m_1, m_2), \phi) = \bigcap_{str \in S} pcheck(merge_G^{str}(m_1, m_2), \phi),$$

with S the set of all valid conflict resolution strategies.

Proof. (Sketch) Follows directly from the fact that $merge_G^{min}(m_1, m_2)$ *is the smallest common subgraph of all graphs produced by the operation merge for any valid conflict resolution strategy.* □

If there are no conflicts in the merged model operations, the *merge* operation produces the same result regardless of the chosen strategy for conflict resolution.

For a correct version history, two model versions M_i and M_j, and the set of versions $P = pre(i) \cap pre(j)$, we define the function

$$pre^C(i,j) = \begin{cases} \emptyset & M_i \in pre(j) \vee M_j \in pre(i) \\ \{M_c \in P \,|\, \forall M_x \in P : M_c \notin pre(x)\} & \text{otherwise} \end{cases},$$

which returns the set of latest common predecessors of M_i and M_j. Note that our definition of pre^C corresponds to the definition of a best common ancestor in conventional version control systems such as Git [16], which is used to compute the base for three-way merges in these systems.

Figure 3 shows an exemplary version history based on the graph M_1 from Fig. 1. The initial graph $M_\alpha = M_1$ contains four classes. The modification m_1 (not to be confused with a morphism) to M_2 creates a superclass edge from c_1 to c_3 and deletes the node c_4. The modification m_2 to graph M_3 creates superclass edges from c_1 to c_2 and from c_4 to c_2. There is an insert-delete conflict between the two modifications, since the modification to M_2 deletes a node that is needed as the source of an edge created by the modification to M_3. Furthermore, the result of the merge of the two modifications would violate the well-formedness constraint with the violation pattern Q from Fig. 1, since without additional modifications, the node c_1 would have two outgoing superclass edges.

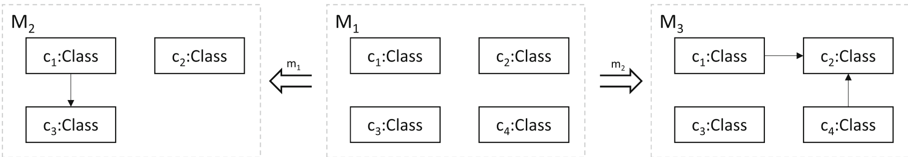

Fig. 3. Example version history

4 Multi-version Models as Typed Graphs

A correct version history $\Delta^{M_{\{1,\ldots,n\}}}$ with model versions $M_{\{1,\ldots,n\}}$ conforming to a type graph TM can be represented by a multi-version model in the form of a single graph that is typed over an adapted type graph.

The adapted type graph TM_{mv} contains a node for each node and edge in TM. It also contains edges connecting each node in TM_{mv} that represents an edge in TM to the nodes representing the edge's source and target in TM. This yields a bijective function $corr_{mv} : V^{TM} \cup E^{TM} \rightarrow V^{TM_{mv}}$, which maps elements from TM to the corresponding node in TM_{mv}, and two bijective functions $corr^s_{mv}, corr^t_{mv} : E^{TM} \rightarrow E^{TM_{mv}}$ mapping edges from TM to the edges in TM_{mv} encoding the source and target relation in TM. In addition, TM_{mv} contains a node *version*, an edge *suc* with source and target *version*, and two edges cv_v and dv_v from each other node $v \in V^{TM_{mv}}$ to the *version* node.

A multi-version model MVM for $\Delta^{M_{\{1,\ldots,n\}}}$ is then constructed by an operation *comb* as follows: A subgraph P^M_{mv} encodes structural information about all model versions and is constructed by translating $P^M = \bigcup_{M_i \in M_{\{1,\ldots,n\}}} M_i$ to conform to TM_{mv} using an operation $trans_{mv}$. Since source and target functions are invariant in a correct version history, P^M is well-defined.

For each $v \in v^{P^M}$, $trans_{mv}$ creates a node of type $corr_{mv}(v)$ in $V^{P^M_{mv}}$. For each $e \in E^{P^M}$, a node of type $corr_{mv}(e)$ is created. This yields a bijection $origin : P^M_{mv} \rightarrow P^M$ mapping translated elements to their original representation.

In addition, for each edge $e \in E^{P^M}$, an edge of type $corr^s_{mv}(e)$ with source $origin^{-1}(e)$ and target $origin^{-1}(s^{P^M}(e))$ and an edge of type $corr^t_{mv}(e)$ with source $origin^{-1}(e)$ and target $origin^{-1}(t^{P^M}(e))$ are created in $E^{P^M_{mv}}$. Since edge sources and targets are invariant, the corresponding node $v_e = origin^{-1}(e)$ in the end has exactly one edge of type $corr^s_{mv}(e)$ and one of type $corr^t_{mv}(e)$. We thus have two functions $s_{mv} : origin^{-1}(E^{P^M}) \rightarrow E^{P^M_{mv}}$ respectively $t_{mv} : origin^{-1}(E^{P^M}) \rightarrow E^{P^M_{mv}}$ encoding these mappings.

Another, distinct subgraph P^V_{mv} contains versioning information and is constructed as follows: For each $M_i \in M_{\{1,\ldots,n\}}$, P^V_{mv} contains a corresponding node of type *version*. For each $(M_i \leftarrow K \rightarrow M_j) \in \Delta^{M_{\{1,\ldots,n\}}}$, P^V_{mv} contains an edge of type *suc* from the node representing M_i to the node representing M_j.

For each modification $(M_i \leftarrow K \rightarrow M_j)$, a *cv*-edge with the node corresponding to M_j as its target is added to all nodes corresponding to elements created by the modification. A *dv*-edge with the node corresponding to M_j as its target is added to all nodes corresponding to elements deleted by the modification. Additionally, a *cv* edge with the node corresponding to the initial version M_α as its target is added to all nodes corresponding to elements in M_α.

Since attributes can be encoded by dedicated nodes and assignment edges [17], the construction can be performed analogously for attributed graphs.

For $v \in P^M_{mv}$ and $M_i \in M_{\{1,\ldots,n\}}$, we say that v is *mv-present* in M_i, iff for a node m_{cv} connected to v via a *cv* edge, there exists a path from m_{cv} to the node representing M_i via *suc* edges that does not go through a node connected to v via a *dv* edge. We denote the set of versions where v is mv-present by $p(v)$.

A model version M_i can then be derived from MVM via an operation *proj* as follows: Collect all nodes $V_p = \{v_p \in V^{P^M_{mv}} | M_i \in p(v_p)\}$, that is, all nodes that are mv-present in M_i, and translate the induced subgraph into the single-version model M_i with $V^{M_i} = \{origin(v_v) | v_v \in V^{MVM} \wedge corr^{-1}_{mv}(type^{MVM}(v_v)) \in V^{TM}\}$, $E^{M_i} = \{origin(v_e) | v_e \in V^{MVM} \wedge corr^{-1}_{mv}(type^{MVM}(v_e)) \in E^{TM}\}$, $s^{M_i} = origin \circ t^{MVM} \circ s_{mv} \circ origin^{-1}$, and $t^{M_i} = origin \circ t^{MVM} \circ t_{mv} \circ origin^{-1}$.

Correctness

Theorem 2. *For a correct version history $\Delta^{M_{\{1,\ldots,n\}}}$ holds concerning comb and proj:*

$$\forall i \in \{1, \ldots, n\} : M_i = proj(comb(\Delta^{M_{\{1,\ldots,n\}}}), i).$$

Proof. (Sketch) Any element in a version M_i has a corresponding node v in $comb(\Delta^{M_{\{1,\ldots,n\}}})$. By construction, v is connected to a node corresponding to some version M_j via a cv edge, for which there exists a path of suc edges to the node corresponding to M_i. That path does not go through a node connected to v by a dv edge. v is thus mv-present in M_i and hence contained in the projection.

Inclusion of elements in the opposite direction can be shown analogously. Because edge sources and targets are invariant over all graphs, the edges in $comb(M_1, \ldots, M_n)$ correctly encode the source and target functions by construction. Thus, $\forall i \in \{1, \ldots, n\} : M_i = proj(comb(M_1, \ldots, M_n), i)$. □

More detailed proofs for this and other theorems in the paper can be found in the appendix of the preprint version [2].

A maximally preserving model modification $(M_i \leftarrow K \rightarrow M_j)$ with $M_i, M_j \in M_{\{1,\ldots,n\}}$ (and thus any model modification in $\Delta^{M_{\{1,\ldots,n\}}}$) can be derived from MVM via $proj^{\Delta}$ as follows: M_i and M_j can be derived via the operation *proj*. K is then the graph containing all elements from $M_i \cap M_j$, with s^K and t^K uniquely defined by the corresponding functions from M_i and M_j and partial identities as morphisms into M_i and M_j.

Theorem 3. *For a correct version history $\Delta^M_{\{1,\ldots,n\}}$ holds concerning comb and $proj^{\Delta}$:*

$$\forall M_i, M_j \in M_{\{1,\ldots,n\}} : m_{i,j} = proj^{\Delta}(comb(\Delta^{M_{\{1,\ldots,n\}}}), i, j),$$

with $m_{i,j}$ the maximally preserving model modification from M_i to M_j.

Proof. Follows trivially from Theorem 2 and the definition of the maximally preserving model modification $(M_i \leftarrow K_{i,j} \rightarrow M_j)$. □

Figures 4 and 5 visualize the multi-version model MVM constructed for the example history in Fig. 3 and the associated adapted type graph TM_{mv}. MVM contains a node for each node and edge in the models of the example history, one node of type *version* for each of the graphs M_1, M_2, and M_3, and appropriate edges as created by *comb*.

4.1 Directly Checking Well-Formedness for Multi-version Models

We can use a multi-version model to directly find all well-formedness violations in all individual versions via an operation $pcheck_{mv}$. For a multi-version model MVM with a bijective mapping into a union of original model versions $origin_M$ and a well-formedness constraint ϕ with associated violation pattern Q, $pcheck_{mv}(MVM, \phi)$ works as follows:

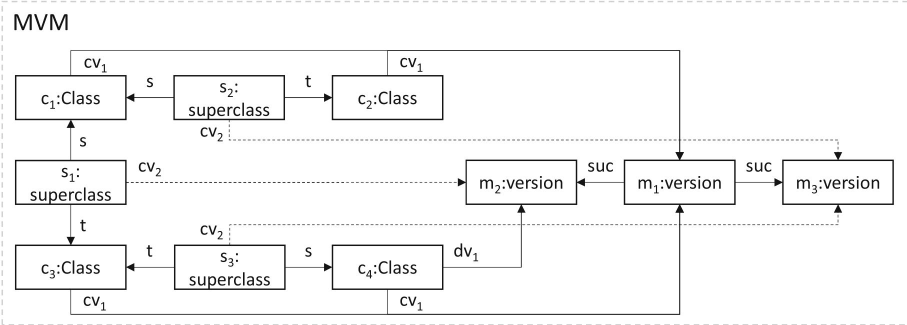

Fig. 4. Multi-version model for the history in Fig. 3

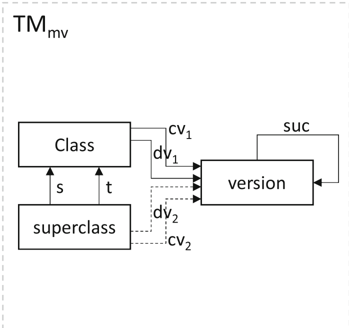

Fig. 5. Adapted type graph for type graph in Fig. 1

First, the graph Q typed over the original type graph is translated into a corresponding graph Q_{mv} typed over the adapted type graph using $trans_{mv}$. This yields a bijective mapping $origin_Q : Q_{mv} \rightarrow Q$.

Then, all matches for Q_{mv} in MVM are found. For each such match m_{mv}, $pcheck_{mv}$ computes all versions for which all vertices in the image of the match are mv-present by $P = \bigcap_{v \in V^{Q_{mv}}} p(m_{mv}(v))$. If $P \neq \emptyset$, the match into the original model versions $m = origin_M \circ m_{mv} \circ origin_Q^{-1}$ is constructed and reported as a violation in all versions in P.

Correctness

Theorem 4. *For a well-formedness constraint ϕ with violation pattern Q, a correct version history $\Delta^{M_{\{1,\ldots,n\}}}$, and $MVM = comb(\Delta^M_{\{1,\ldots,n\}})$ holds:*

$$pcheck_{mv}(MVM, \phi) = \biguplus_{i \in \{1,\ldots,n\}} \{(i, m) | m \in pcheck(proj(MVM, i), \phi)\}.$$

Proof. (Sketch) A match $m : Q \rightarrow M_i$ for any version M_i has one corresponding match m_{mv} with $m = origin_M \circ m_{mv} \circ origin_Q^{-1}$, where edges created by

$trans_{mv}$ ensure correct connectivity. $P = \bigcap_{v \in V^{Q_{mv}}} p(m_{mv}(v))$ contains exactly the versions containing all elements in $m(Q)$. This yields the stated equality. $\square$

Complexity. The effort for searching all versions $M_{\{1,\ldots,n\}}$ of some version history $\Delta^{M_{\{1,\ldots,n\}}}$ for a pattern Q using *pcheck* is in $O(\sum_{M_i \in M_{\{1,\ldots,n\}}} C(M_i, Q))$, with $C(M_i, Q)$ the effort for finding all matches of Q into M_i.

$P^M_{mv} = trans_{mv}(P^M)$ and $Q_{mv} = trans_{mv}(Q)$ are only different encodings of $P^M = \bigcup_{M_i \in M_{\{1,\ldots,n\}}} M_i$ and Q. Considering computation of the mv-present predicate, the effort for $pcheck_{mv}$ is hence in $O(C(\bigcup_{M_i \in M_{\{1,\ldots,n\}}} M_i, Q) + X \cdot |V^{Q_{mv}}| \cdot |\Delta^{M_{1,\ldots,n}}|)$, with X the number of matches for Q_{mv} into P^M_{mv}.

Discussion. If many elements are shared between individual versions and modifications only perform few changes, the size of the union of all model versions will be small compared to the sum of the sizes of all individual versions. If pattern matching is efficient with respect to the size of the considered model, pattern matching over the union of all model versions will then likely require less effort than matching over each individual version. Intuitively, $pcheck_{mv}$ avoids redundant searches over model parts that are shared between multiple versions and thus saves the related effort. If the number of matches for violation patterns is low, the associated checks performed by $pcheck_{mv}$ will likely be more efficient than the pattern matching over the individual versions.

Overall, $pcheck_{mv}$ will thus likely be more efficient than using *pcheck* in scenarios where pattern matching is efficient, the number of changes between versions is low, and the number of violations in the union of versions is low.

5 Directly Checking Merge Results for Multi-version Models

We can consider multi-version models to directly detect whether (a) merge conflicts exist for any valid pair of encoded model modifications via an operation $mcheck_{mv}$ and (b) any resulting merged model is ill-formed via an operation $pcheck^m_{mv}$, where a pair of model modifications $(M_c \leftarrow K_i \rightarrow M_i)$ and $(M_c \leftarrow K_j \rightarrow M_j)$ is valid iff $M_c \in pre^C(M_i, M_j)$.

5.1 Directly Checking for Merge Conflicts

$mcheck_{mv}$ can be realized for a multi-version model $MVM = comb(\Delta^{M_{\{1,\ldots,n\}}})$ as follows: First, the operation collects all nodes in MVM representing edges that are created by some model modification. This means all nodes $v_e \in V^{MVM}$ where $corr^{-1}_{mv}(type^{MVM}(v)) \in E^{TM}$ connected to a node m_x via a *cv* edge, where m_x does not correspond to M_α and with TM the original type graph. Then, for each node v_e, we compute the set of versions $P = p(v_e)$ where it is mv-present. If $P \neq p(v_s)$, where $v_s = s^{MVM}(s_{mv}(v_e))$, we then compute a set of versions D

that correspond to nodes reachable via *suc* edges from a node connected to v_s via a *dv* edge without going through nodes connected to v_s via a *cv* edge.

Afterwards, for each pair of versions $M_i \in P$ and $M_j \in D$, we check for each latest common predecessor $M_c \in pre^C(i,j)$ whether $M_c \in p(v_s) \wedge M_c \notin P$. For any triplet of versions (i,j,c) where this is the case, the edge $origin(v_e)$ is then in an insert-delete conflict with its source. To facilitate formalization, this conflict is reported in the normalized form $(min(i,j), max(i,j), c, (origin(v_e), origin(v_s)))$. Insert-delete conflicts with the edge's target are computed analogously.

Correctness

Theorem 5. *For a version history $\Delta^{M_{\{1,\ldots,n\}}}$ and the associated multi-version model $MVM = comb(\Delta^M_{\{1,\ldots,n\}})$ holds:*

$$mcheck_{mv}(MVM) = \biguplus_{(i,j,c) \in Y} \{(i,j,c,m) | m \in mcheck(m_{c,i}, m_{c,j})\},$$

where $Y = \{(i,j,c) \,|\, i,j \in \{1,\ldots,n\} : i < j, c \in \{c | M_c \in pre^C(i,j)\}\}$ and with $m_{c,i} = proj^\Delta(MVM, c, i)$ and $m_{c,j} = proj^\Delta(MVM, c, j)$.

Proof. (Sketch) The collected nodes representing edges correspond to a superset of edges that may be involved in a conflict. The construction of the sets P and D for a collected node v_e ensures that any pair of versions where one may create $e = origin(v_e)$ and the other may delete the source (or target) of e is considered. The condition checked for each common predecessor of a version pair then yields exactly the triplets of versions where e is part of an insert-delete conflict. Because of the normalization of the results of $mcheck_{mv}$, we have the stated equality. □

Complexity. The function pre^C_{mv} can be precomputed in $O(|M_{\{1,\ldots,n\}}|^4)$.

Since information about creation and deletion of elements is not explicitly available in a naïve representation, finding all insert-delete conflicts between two model modifications via *mcheck* has to be done by checking for each edge in either modification's resulting model whether it is created by that modification and its source or target is deleted by the other modification. Since there may exist up to $O(|M_{\{1,\ldots,n\}}|^3)$ possible merges in a version history, in the worst case, this implies effort in $O(|M_{\{1,\ldots,n\}}|^4 + |E^{M_{max}}| \cdot |M_{\{1,\ldots,n\}}|^3)$, where $|E^{M_{max}}|$ is the maximum number of edges present in a single model version.

Created edges can be retrieved efficiently from a multi-version model given appropriate data structures. Computing and checking the required version sets takes $O(|M_{\{1,\ldots,n\}}|^3)$ steps per edge. Therefore, the overall computational complexity of $mcheck_{mv}$ is in $O(|M_{\{1,\ldots,n\}}|^4 + \Delta_+ \cdot |M_{\{1,\ldots,n\}}|^3)$, where Δ_+ is the overall number of elements created in the version history.

Discussion. The efficiency of $mcheck_{mv}$ compared to using $mcheck$ mostly depends on the number of edges created by some model modification compared to the number of edges in the individual versions. If most edges are present in the original model version and are shared between many model versions, $mcheck_{mv}$ will be more efficient. Otherwise, $mcheck_{mv}$ will not achieve a significant improvement and might even perform worse than the operation based on $mcheck$.

Version control systems such as Git typically select a single latest common predecessor as the base for a three way merge [16]. Using a corresponding partial function $pre^C_1 : \mathbb{N} \times \mathbb{N} \rightarrow M_{\{1,\ldots,n\}}$ with $pre^C_1(i,j) \in pre^C(i,j)$ if $pre^C(i,j) \neq \emptyset$ and $pre^C_1(i,j) = \perp$ to select a single latest common predecessor of two versions i and j rather than pre^C in $mcheck_{mv}$, by the same logic as used in the proof of correctness, we instead have an analogous equality for pre^C_1. Disregarding the computational effort for precomputing pre^C_1, replacing pre^C by pre^C_1 reduces the remaining computational complexity of $mcheck_{mv}$ to $O(\Delta_+ \cdot |M_{\{1,\ldots,n\}}|^2)$.

5.2 Directly Checking Well-Formedness for Merge Results

To find all violations of a well-formedness constraint ϕ characterized by a pattern Q via $pcheck^m_{mv}$ in merge results of a multi-version model MVM, we first translate Q into $Q_{mv} = trans_{mv}$. We then find all matches for Q_{mv} in MVM.

For a match m_{mv} for Q_{mv}, we determine the set of versions $P_v = p(v)$ for each $v \in m_{mv}(V^{Q_{mv}})$. For each pair of versions $M_i \in \arg\min_{P \in \{p(v) | v \in m_{mv}(V^{Q_{mv}}\}} |P|$ and $M_j \in \bigcup_{v \in V^{Q_{mv}}} p(v)$, we check whether $\forall v \in m_{mv}(V^{Q_{mv}}) : M_i \in p(v) \vee M_j \in p(v)$. We then check for each latest common predecessor $M_c \in pre^C(i,j)$ if for all $v \in V^{Q_{mv}}$, it holds that $v \in V^{M_c} \rightarrow (v \in V^{M_i} \wedge v \in V^{M_j})$, that is, v is not deleted in M_i or M_j. If this is the case, the match m into $\bigcup_{M_x \in M_{\{1,\ldots,n\}}} M_x$ corresponding to m_{mv} represents a violation in $merge^{min}((M_c \leftarrow K_i \rightarrow M_i), (M_c \leftarrow K_j \rightarrow M_j))$. We report results in the normalized form $(min(i,j), max(i,j), c, m)$.

Correctness

Theorem 6. *Given a well-formedness constraint ϕ, a correct version history $\Delta^{M_{\{1,\ldots,n\}}}$, and the multi-version model $MVM = comb(\Delta^M_{\{1,\ldots,n\}})$, it holds that:*

$$pcheck^m_{mv}(MVM, \phi) = \biguplus_{(i,j,c) \in Y} \{(i,j,c,m) | m \in pcheck(M^{min}_{i,j,c}, \phi)\},$$

where $Y = \{(i,j,c) \,|\, i,j \in \{1,\ldots,n\} : i < j, c \in \{c | M_c \in pre^C(i,j)\}\}$ and $M^{min}_{i,j,c} = merge^{min}_G(proj^\Delta(MVM,c,i), proj^\Delta(MVM,c,j))$.

Proof. (Sketch) For two versions M_i, M_j with latest common predecessor M_c, a match $m : Q \rightarrow merge^{min}_G(proj^\Delta(MVM,c,i), proj^\Delta(MVM,c,j))$ has one corresponding match $m_{mv} : trans_{mv}(Q) \rightarrow MVM$ by construction, where the edges created by $trans_{mv}$ ensure the correct connectivity. The set of version pairs considered by $pcheck^m_{mv}$ contains all version pairs such that each matched element *is contained in at least one of the versions. The condition checked for every latest common predecessor ensures that only version triplets are reported where the merge result also contains all matched elements if there are no merge conflicts. Since* $merge^{min}$ *resolves conflicts by prioritizing deletion and, as ensured by the check, no matched node is deleted by the merge, conflict resolution cannot invalidate the match or create new matches. We thus have the stated equality.* □

By Theorem 1 and Theorem 6, we also have that $pcheck^m_{mv}$ yields the set of violations that cannot be avoided by any conflict resolution strategy:

Corollary 1. *Given a well-formedness constraint* ϕ*, a correct version history* $\Delta^{M_{\{1,\dots,n\}}}$*, and the multi-version model* $MVM = comb(\Delta^{M_{\{1,\dots,n\}}})$*, it holds that:*

$$pcheck^m_{mv}(MVM, \phi) = \biguplus_{(i,j,c)\in Y} \bigcap_{str\in S} \{(i,j,c,m) | m \in mcheck(M^{str}_{i,j,c}, \phi)\},$$

where $Y = \{(i,j,c) \,|\, i,j \in \{1,\dots,n\} : i < j, c \in \{c | M_c \in pre^C(i,j)\}\}$ *and* $M^{str}_{i,j,c} = merge^{str}_G(proj^\Delta(MVM,c,i), proj^\Delta(MVM,c,j))$*, and with* S *the set of all valid conflict resolution strategies.*

Complexity. The function pre^C_{mv} can be precomputed in $O(|M_{\{1,\dots,n\}}|^4)$.

With $C(M_i, Q)$ the effort for finding all matches of Q into M_i, finding violations characterized by a pattern Q in all results of a set of possible merges Y using *pcheck* takes effort in $O(O(|M_{\{1,\dots,n\}}|^4 + \sum_{(m_1,m_2)\in Y} C(merge^{min}_G(m_1,m_2), Q))$.

The computation and checking of version triplets for a match in $pcheck^m_{mv}$ takes effort in $O(|M_{\{1,\dots,n\}}|^3)$. For X matches for Q_{mv}, the effort for $pcheck^m_{mv}$ is thus in $O(|M_{\{1,\dots,n\}}|^4 + C(\bigcup_{M_i\in M_{\{1,\dots,n\}}} M_i, Q) + X \cdot |V^{Q_{mv}}| \cdot |M_{\{1,\dots,n\}}|^3)$.

Discussion. By the same argumentation as for $pcheck_{mv}$, $pcheck^m_{mv}$ will likely be more efficient than the corresponding operation using *pcheck* in scenarios where pattern matching is efficient, the number of changes between versions is low, and the number of violations in the union of model versions is low.

Using some partial function $pre^C_1 : \mathbb{N} \times \mathbb{N} \to M_{\{1,\dots,n\}}$ to select a single latest common predecessor rather than pre^C in $pcheck^m_{mv}$, by the same logic as in the proof of correctness, we have an analogous equality for pre^C_1. Disregarding the effort for precomputing pre^C_1, replacing pre^C by pre^C_1 reduces the remaining complexity of $pcheck^m_{mv}$ to $O(C(\bigcup_{M_i\in M_{\{1,\dots,n\}}} M_i, Q) + X \cdot |V^{Q_{mv}}| \cdot |M_{\{1,\dots,n\}}|^2)$.

6 Evaluation

For an initial empirical evaluation of the performance and scalability of the presented operations, we experiment with an application scenario from the software development domain. Therefore, we extract abstract syntax graphs from a small previous research project (**rete**) and a larger open source project (**henshin** [1]) written in Java using the EMF-based [10] MoDisco tool [5]. We store the extracted models in a graph format and fold each of the projects into a multi-version model, using a mapping strategy based on hierarchy and element names.

We then run implementations of the presented operations for conflict detection and well-formedness checking based on multi-version models (**MVM**) and baseline implementations using corresponding single-version models (**SVM**).[1] We consider three well-formedness constraints: uniqueness of a class's superclass, uniqueness of a method's return type, and consistency of an overriden method's return type. We employ our own EMF-based tool [14] for pattern matching.

Figure 6 shows the measured execution times for the operations $pcheck_{mv}$, $mcheck_{mv}$, and $pcheck^m_{mv}$ and related single-version-model-based operations over the example models. The execution times for $pcheck_{mv}$ and $pcheck^m_{mv}$ correspond to the combined pattern matching time for all considered well-formedness constraints. All reported times exclude the time for computing any merge results required by SVM and the time required to precompute the pre^C function, since it is required by both the MVM and the SVM implementation. Precomputing pre^C took about 5 ms for the smaller project and about 3.5 s for the larger project.

For the tasks related to well-formedness checking, the MVM variant performs better (up to factor 50) than SVM. Since there are only few to no matches for the violation patterns of the considered constraints, the MVM implementation only performs few of the potentially expensive checks over the version graph, while avoiding most of the redundancy in the pattern matching of SVM.

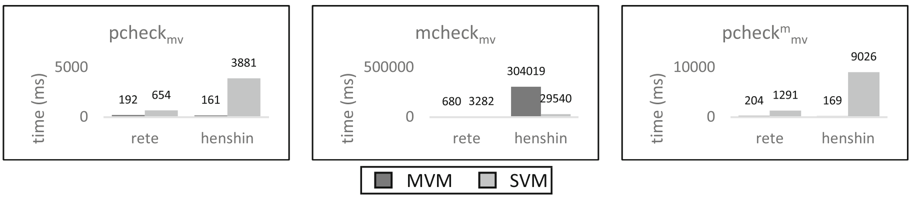

Fig. 6. Measurement results for $pcheck_{mv}$, $mcheck_{mv}$, and $pcheck^m_{mv}$

For conflict detection, MVM performs better than SVM for the smaller project (factor 5), but has a substantially higher execution time for the larger project (factor 10). The reason for the bad performance is that most edges are not present in the initial model version. In fact, the number of edges created throughout the version history is much higher than the number of edges in any individual version. Furthermore, in contrast to the solution using *mcheck*, the operation $mcheck_{mv}$ considers versions where the source or target of an edge

[1] All experiments were executed on a Linux SMP Debian 4.19.67-2 machine with Intel Xeon E5-2630 CPU (2.3 GHz clock rate) and 386 GB system memory running OpenJDK version 1.8.0_242. Reported execution times correspond to the minimum of at least five runs of the respective experiment. Memory measurements were obtained in a single run using the native Java library. Our implementation and datasets are available under https://github.com/hpi-sam/multi-version-models.

is *not* present. Due to the high number of versions in the project and because many elements are only present in few versions, this leads to the processing of large version sets, which deteriorates the performance of MVM in this scenario.

The memory consumption of the multi-version models and their representations as collections of single-version models is displayed in Fig. 7. For both projects, the representation as a multi-version model affords a more compact representation compared to a naïve encoding (factor 30 for the larger project).

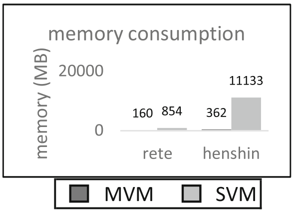

Fig. 7. Measurement results for memory consumption

Threats to Validity. Unexpected JVM behavior poses a threat to internal validity, which we tried to mitigate by performing multiple runs of each experiment measuring execution time and profiling time spent on garbage collection. To address threats to external validity, we used real-world data and well-formedness constraints in our experiments. While we used our own tool for pattern matching, said tool has already been used in our previous works and has shown adequate performance [14].

However, the example constraints are not representative and the folding of individual model versions extracted from source code may yield a larger-than-necessary multi-version model. Our results are thus not necessarily generalizable, but instead constitute an early conceptual evaluation of the presented approach.

7 Related Work

While most practical version control systems operate on text documents [20], versioning and merging of models has also been subject to extensive research.

There already exist several formal and semi-formal approaches to model merging, which compute the result of a three-way-merge of model modifications [26,27]. Notably, the approach by Taentzer et al. [26] represents a formally defined solution that works on the level of graphs, which is why for our approach, we build on their notion of model merging. In their work, Taentzer et al. also consider checking of well-formedness constraints by constructing a tentative merge result over which the check is executed. While this allows their approach to handle arbitrary constraints rather than just simple graph patterns, the check has to be executed for each individual merge.

Some approaches consider detection of merge conflicts [19] or model inconsistencies [3] based on the analysis of sequences of primitive changes. However, these approaches do not consider the case of multiple versions and pairwise merges and naturally do not employ a graph-based definition of inconsistencies.

For the more general problem of model versioning, both formal solutions [8,24] and tool implementations [18,21] have been introduced. Similar to our approach, some of these techniques are based on a joint representation of multiple model versions [21,24]. However, to the best of our knowledge, joint conflict detection or well-formedness checking for all merges at once is not considered.

Model repositories such as Hawk [13] allow storing the evolution of models over time and enable the execution of queries equipped with temporal operators. Folding and joint querying of the temporal evolution of graphs has also been studied in previous work of our group [15,25]. However, these solutions focus on sequences of graph modifications without diverging branches and hence do not consider merging.

The presented encoding of different model versions in a unified multi-version model bears similarity to so-called 150% models from software product lines [6]. A 150% model represents different configurations of a software system as a single unified model, where annotations determine the presence of individual model elements in certain configurations. The derivation of a model instance for a specific configuration from a 150% model then corresponds to the projection from a multi-version model to a specific model version. A realization of 150% models in the context of model-driven engineering is presented in [23].

Westfechtel and Greiner [28] present a solution for propagating presence information from a unified encoding of multiple product line configurations along model transformations. While their approach bears some similarity to the collective well-formedness checking in our solution, the technique in [28] focuses on product lines and hence does not consider version histories and merging.

[7] introduces a new semantics for OCL in the context of software product lines, which allows the collective checking of well-formedness constraints over a unified encoding of product line configurations. However, the application of this approach to model versioning would require a translation of version graphs and model modifications to an encoding of valid configurations and presence annotations. This seems nontrivial, especially if the compression of version histories achieved by multi-version models is to be preserved. However, by relying on OCL as a specification language, the approach in [7] allows a much higher expressiveness when formulating well-formedness conditions compared to simple graph patterns. Adopting some of the ideas in [7] may therefore enable lifting our definition of well-formedness to more expressive formalisms in future work.

A solution to conflict detection for features in software product lines is presented in [22]. In [22], product variability is encoded by so-called delta modules, which represent operations for extending a basic version of the software by certain features and are thus similar to model modifications. The approach checks for syntactic conflicts via pair-wise comparison of delta-modules and thus relates to detection of merge conflicts in the context of model merging. The approach in [22] also considers the case where a third delta module fixes conflicts between two other modules. Considering merges of more than two versions could also be an interesting direction for future work in the context of multi-version models.

8 Conclusion

In this paper, we have presented an approach for encoding a model's version history as a single typed graph. Based on this representation, we have introduced operations for finding merge conflicts and violations of well-formedness conditions in the form of graph patterns in the entire history and related merge results. We have conducted an initial empirical evaluation, which demonstrates potential benefits of the approach, but also highlights shortcomings in unfavorable scenarios.

In future work, we plan to address these shortcomings by studying how to compress the version graph or restrict the set of considered versions to those most relevant to users. We also plan to explore how such a restriction may allow the pruning of superfluous elements from a multi-version model and thereby prevent performance degradation as more versions are introduced. Furthermore, we want to investigate how to lift our notion of well-formedness constraints to more expressive formalisms such as nested graph conditions and develop an incremental version of the approach. Finally, we will extend our empirical evaluation to better characterize our technique's performance.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9
2. Barkowsky, M., Giese, H.: Towards Development with Multi-Version Models: Detecting Merge Conflicts and Checking Well-Formedness. arXiv preprint (2022). https://doi.org/10.48550/arXiv.2205.04198
3. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Incremental detection of model inconsistencies based on model operations. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 32–46. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02144-2_8
4. Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Early detection of collaboration conflicts and risks. IEEE Trans. Softw. Eng. **39**(10), 1358–1375 (2013). https://doi.org/10.1109/TSE.2013.28
5. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (2010). https://doi.org/10.1145/1858996.1859032
6. Czarnecki, K., Antkiewicz, M.: Mapping features to models: a template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005). https://doi.org/10.1007/11561347_28

7. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 211–220. Association for Computing Machinery, New York (2006). https://doi.org/10.1145/1173706.1173738
8. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: algebraic foundations and the tile notation. In: 2009 ICSE Workshop on Comparison and Versioning of Software Models, pp. 7–12. IEEE (2009). https://doi.org/10.1109/CVSM.2009.5071715
9. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS, Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
10. EMF. https://www.eclipse.org/modeling/emf/. Accessed 23 Feb 2022
11. Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multiperspective specifications. IEEE Trans. Software Eng. **20**(8), 569–578 (1994). https://doi.org/10.1109/32.3106670
12. Frühauf, K., Zeller, A.: Software configuration management: state of the art, state of the practice. In: SCM 1999. LNCS, vol. 1675, pp. 217–227. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48253-9_15
13. García-Domínguez, A., Bencomo, N., Parra-Ullauri, J.M., García-Paucar, L.H.: Querying and annotating model histories with time-aware patterns. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 194–204. IEEE (2019). https://doi.org/10.1109/MODELS.2019.000-2
14. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Electronic Communications of the EASST, vol. 18 (2009). https://doi.org/10.14279/tuj.eceasst.18.268
15. Giese, H., Maximova, M., Sakizloglou, L., Schneider, S.: Metric temporal graph logic over typed attributed graphs. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 282–298. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_16
16. Git. https://git-scm.com/. Accessed 23 Feb 2022
17. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45832-8_14
18. Koegel, M., Helming, J.: EMFStore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, pp. 307–308 (2010). https://doi.org/10.1145/1810295.1810364
19. Küster, J.M., Gerth, C., Engels, G.: Dependent and conflicting change operations of process models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 158–173. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02674-4_12
20. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. **28**(5), 449–462 (2002). https://doi.org/10.1109/TSE.2002.1000449
21. Murta, L., Corrêa, C., Prudêncio, J.G., Werner, C.: Towards odyssey-VCS 2: improvements over a UML-based version control system. In: Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models (2008). https://doi.org/10.1145/1370152.1370159

22. Pietsch, C., Kelter, U., Kehrer, T.: From pairwise to family-based generic analysis of delta-oriented model-based SPLs, pp. 13–24. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3461001.3471150
23. Reuling, D., Pietsch, C., Kelter, U., Kehrer, T.: Towards projectional editing for model-based SPLs. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems, VAMOS 2020. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3377024.3377030
24. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A category-theoretical approach to the formalisation of version control in MDE. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 64–78. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_5
25. Sakizloglou, L., Ghahremani, S., Barkowsky, M., Giese, H.: Incremental execution of temporal graph queries over runtime models with history and its applications. Softw. Syst. Model. 1–41 (2021). https://doi.org/10.1007/s10270-021-00950-6
26. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: A fundamental approach to model versioning based on graph modifications: from theory to implementation. Softw. Syst. Model. **13**(1), 239–272 (2012). https://doi.org/10.1007/s10270-012-0248-x
27. Westfechtel, B.: A formal approach to three-way merging of EMF models. In: Proceedings of the 1st International Workshop on Model Comparison in Practice (2010). https://doi.org/10.1145/1826147.1826155
28. Westfechtel, B., Greiner, S.: Extending single- to multi-variant model transformations by trace-based propagation of variability annotations. Softw. Syst. Model. **19**(4), 853–888 (2020). https://doi.org/10.1007/s10270-020-00791-9