Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

From Imprecise N-Way Model Matching to Precise N-Way Model Merging

Dennis Reuling[a] Malte Lochau[b] Udo Kelter[a]

a. Software Engineering Group, University of Siegen, Germany

b. Real-Time Systems Lab, TU Darmstadt, Germany

Abstract

N-way model merging is a key technique for managing software variability, by integrating N concurrent model variants/versions into one. Most merging techniques are based on three operators: (1) compare, (2) match and (3) merge. As finding optimal solutions for (2) in terms of matching precision is NP-hard, many proposals for scalable yet sufficiently precise matching heuristics exist. However, most approaches are either generic which obstructs precision if applied to realistic modeling languages, or they require excessive computational upfront investment already in step (1) to ensure sufficient precision. We propose an alternative approach for automated construction of precise model merges by focusing on step (3): given an arbitrarily imprecise matching, we incrementally apply default model-refactoring operators, as available for most mature modeling languages, to identify and unify further similarities among initially unmatched model elements. For those model-refactoring operators to produce correct results if applied to merged models, we utilize variability encoding as known from product-line engineering. Our tool implementation supports any EMF-compliant modeling language and is instantiated for UML class diagrams to demonstrate our methodology. Our evaluation results show that our technique (a) preserves the precision of near optimal matchings and (b) remarkably improves the precision of arbitrarily imprecise matchings while requiring acceptable computational effort.

Keywords Model Merging, Model Transformation, Software Variability

1 Introduction

In many modern application domains, software exists in many variants and/or versions. Model-based engineering offers elaborated techniques and tools to cope with the ever-growing complexity of software due to this inherent *variability*. N-way model merging has become a key technique for variability management, aiming at integrating N often concurrently developed model variants/versions of the same software into one unified representation [RC13a]. Classical merge-operators from repository systems like SVN or GIT usually perform 3-way merging, by actually merging two concurrent variants, whereas a third variant serves as base model for automating merge decisions in case of local differences between variants [Men02]. If a merge decision cannot be made automatically, a *merge conflict* occurs which requires manual resolution [WLS+12]. Obviously, those techniques are insufficient for automated N-way merging of complex graph-based models with rich semantics (e.g., UML class diagrams). Most recent approaches for N-way model merging use a combination of three steps/operators: (1) compare, (2) match, and (3) merge [RC13a]. The computational problem underlying (2) of finding optimal matchings covering a maximum number of similar elements between N input models w.r.t. compare metrics of (1) is known to be NP-hard [RC13a]. Many heuristics have been proposed for computing sufficiently *precise* N-way matchings under reasonable effort [WWS+17, HWL+14, SRA+16, MZB+15]. Unfortunately, these approaches are either generic which may, again, compromise the precision when adapted to realistic modeling languages, or they require excessive computational upfront investment already during step (1) to ensure sufficient precision in step (2). Furthermore, after step (2), most approaches devise a basic unify-merge in step (3) by simply integrating all matched model parts in a "copy-as-is" fashion into one merged model. Unfortunately, this naive approach often fails to produce useful representations as it potentially yields syntactically ill-formed and/or semantically incorrect output models (e.g., in case of merge conflicts). In contrast, lifting off-the-shelf tools for automated software analysis to product-line engineering [TAK+14] usually requires merged models to be at least syntactically well-formed and semantically sound [RKBL19].

In this paper, we propose a novel approach for the automated construction of syntactically well-formed and semantically correct, yet precise N-way model merges incorporating integrated handling of merge conflicts. To this end, we focus on step (3): given an arbitrarily (im-)precise matching as well as syntactically valid input models, our framework applies a catalog of model-transformation operators for fully-automated, yet preferably precise merged representations of all N model variants/versions. For this purpose, our methodology utilizes default *model-refactoring operators*, as available for most mature modeling languages, e.g., UML [SPLTJ01, TS14], EMF [BEK+07], BPMN [WR08] or FODA feature diagrams [SBRCT08a], for identifying and unifying further similarities among initially unmatched model elements in a merged model. For model-refactoring operators to produce correct results if applied to unify-merged models, we utilize the concept of *variability encoding* from product-line engineering [PS08, vRTS+16] to embed variability-information, including conflicting parts, into merged models. To summarize, we make the following contributions.

- We propose a methodology for automated N-way model merging which does not depend on the precision of a given matching. Our framework combines variability-encoding and model-refactoring operators to ensure correctness and to improve precision of the merged model.
- We present a tool supporting any EMF-compliant modeling language and we instantiate our methodology for UML class diagrams.
- We provide evaluation results showing that our approach (a) preserves precision of near optimal matchings and (b) remarkably improves precision of imprecise

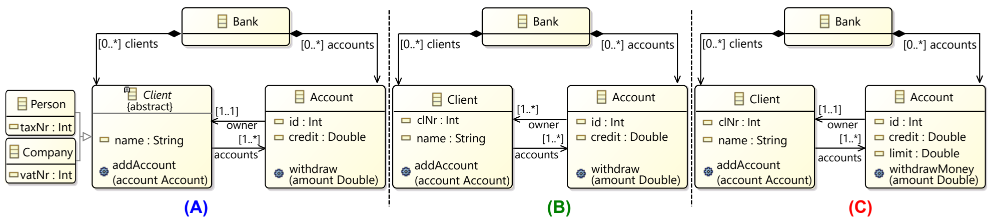


Figure 1 – Class-Diagram Variants of a Banking System.

matchings, both under acceptable computational effort.

We provide our tool and all experiment data on a supplementary web page [acc19].

2 Background and Motivation

Consider the UML class diagrams, *(A)*, *(B)* and *(C)* in Fig. 1, each constituting a variant of a structural specification of a (simplified) banking system [RC13b, ABJ$^+$10, MZB$^+$15, DRB$^+$13, RCC15].

Class Diagram (A). This variant consists of the classes *Bank*, *Client* and *Account*. Each *Client* has an attribute *name* and owns at least one *Account*, where each *Account* is owned by exactly one *Client*. A *Client* is either a single *Person* or a *Company* both identified by their attributes *taxNr* and *vatNr*. Each *Account* consists of a unique *id* and a current *credit* value. Method *addAccount* of class *Client* adds further accounts to a *Client* and a *Client* can *withdraw* money from his/her *Accounts*.

Class diagram (B). In this variant, the classes *Person* and *Company* with their attributes *taxNr* and *vatNr* are replaced by the attribute *clNr* for identifying clients and class *Client* is no longer *abstract*. In addition, this variant permits multiple *Clients* to own the same *Account*.

Class diagram (C). Here, class *Account* owns a further attribute *limit* for overdrawing and method *withdraw* has been renamed to *withdrawMoney*.

Semantically, class diagrams specify (potentially infinite) sets of *valid instantiations* (object diagrams) satisfying all constraints (e.g., cardinality bounds). Due to similarities among *(A)*, *(B)* and *(*C), some instantiations are equally valid for more than one variant, whereas others are only valid for one particular variant.

N-Way Model Merging. The goal of N-way model merging is to make explicit common and variable parts in a given set of N model variants/versions $m_1, m_2, \ldots m_N$ by constructing one *merged model* $m_{\langle 1..N\rangle}$ which contains all input models and (syntactically) unifies *similar* parts among these models [RC13a]. Most (semi-)automated model-merging approaches are decomposed into three consecutive steps and respective operators called *compare*, *match* and *merge* [RC13a]. The *compare*-step consists of metrics for measuring the *similarity* of (subsets of) model elements from different input models using model-type-specific operators. By E_i, $1 \leq i \leq N$, we denote the sets of elements of the input models m_i, $1 \leq i \leq N$. A *Matching* M for a set of input models $m_1, m_2, \ldots m_N$ then consists of a set of N*-tuples* $T \in M$ with $1 \leq |T| \leq N$ grouping similar model elements such that (1) every element $e_i \in E_i$ of each input model m_i, $1 \leq i \leq N$, is contained in exactly one tuple $T \in M$ and (2) each tuple $T \in M$ contains at most one element from each input model. Hence, M constitutes a *partitioning* of the union of the sets of model elements of all N input models.

In our running example, all classes and respective class members having equal names in *(A)*, *(B)* and *(C)* may be matched into the same tuples, whereas elements with variant-specific elements (e.g., class *Person* and *Company*, attribute *limit*) are matched as dissimilar into singleton tuples. To match corresponding elements even after small changes (e.g., association *owner* or renamed method *withdraw*), most practical approaches [MZB+15, WWS+17] (also) apply so-called identity-based matching [KKPS12], where compare-metrics utilize (internal) identifiers of model elements (e.g., as used by modeling tools). Besides name-based and identity-based matching of major model elements like classes, *compare*-operators for dependent model elements like associations further take context-specific information into account (e.g., similarity of source- and target-classes).

Matching Precision vs. Merge Precision. An optimal (i.e., maximally precise) matching M groups a maximum number of similar elements (w.r.t. the compare-measures) shared between all input models. Computing such an *optimal matching* M (semi-)automatically is known to be NP-hard [RC13a] in the number N of variants under consideration and many heuristics have been developed for sufficiently precise, yet effectively computable N-way matchings [WWS+17, HWL+14, SRA+16]. Unfortunately, these approaches are either generic which may, again, compromise precision when adapted to realistic modeling languages, or they require excessive computational upfront investment already during the compare-step.

Concerning the merge-step, *precision* of the merged models is crucial in various contexts and use cases, e.g., for understanding common and variable parts [EMCLGP09] or (automated) software analyses [TAKS14, BLL+13]. In this regard, different precision criteria may be considered for guiding the merging process. Typically, these criteria use syntactic model metrics which quantify the succinctness of the merge result [RC13b, RKBL19]. We assume, without loss of generality, that one key goal of N-way model merging is to reduce duplications among the merged variants within the merged model. To this end, unifying similar elements (w.r.t. compare-measures) as much as possible potentially leads to a *optimally precise* merged model $m_{\langle 1..N\rangle}$ (i.e., model elements grouped as similar in an (optimal) matching M are, as far as possible, also integrated as one unified element into $m_{\langle 1..N\rangle}$) [RC13b]. We may characterize *precision* of a merged model $m_{\langle 1..N\rangle}$ with respect to any given (i.e., arbitrarily (im-)precise) matching M as

$$Prec(M, m_{\langle 1..N\rangle}) = \frac{|M|}{|E_{\langle 1..N\rangle}|},$$

by relating the number of tuples in M to the resulting number of elements in the merged model $m_{\langle 1..N\rangle}$. Hence, in a precision-preserving merged model $m_{\langle 1..N\rangle}$ for a given matching M, every tuple $T \in M$ is integrated into $m_{\langle 1..N\rangle}$ by exactly one element $e_T \in E_{\langle 1..N\rangle}$.

Concerning our example, if we count class members, i.e., attributes, associations and methods (e.g., Account.id, Bank.clients and Client.addAccount) as model elements then our variants *(A)*, *(B)* and *(C)* consist of $|E_A| = 11$, $|E_B| = 10$ and $|E_C| = 11$ elements, respectively. Figure 2 (on the top) depicts a maximally precise merged model $m_{pr\langle A,B,C\rangle}$ including all elements of *(A)*, *(B)* and *(C)* with $|M| = 13$ tuples, where colors are used to relate elements to variants. Hence, Fig. 2 (on the top) constitutes a precision-preserving merged model for an optimal matching M with $\frac{|M|}{|E_{mpr\langle A,B,C\rangle}|} = \frac{13}{13} = 1.0$, resulting from simply *unifying* each matched element in M into one element in $m_{pr\langle A,B,C\rangle}$. However, this model does not constitute a *correct*

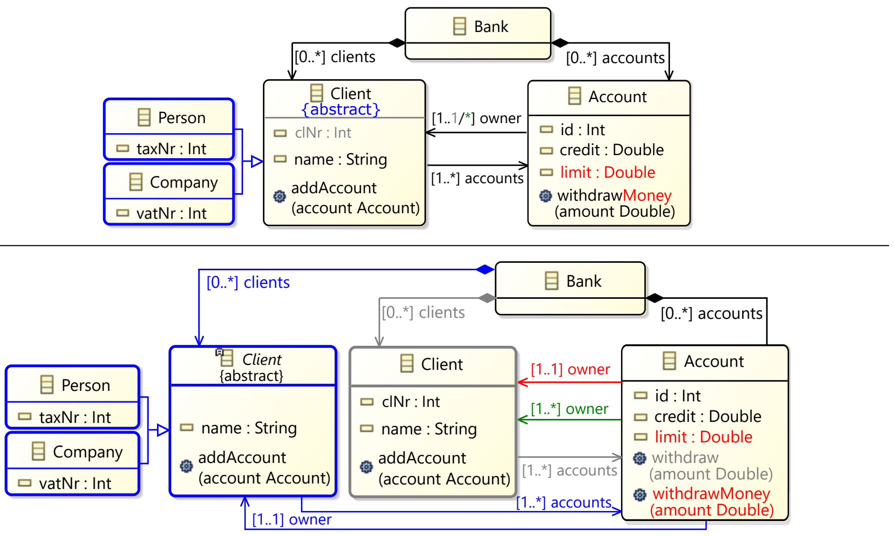


Figure 2 – Precise (top, $m_{pr\langle A,B,C\rangle}$) and Imprecise (bottom, $m_{im\langle A,B,C\rangle}$) N-way unify-merged model of all three variants. Parts common to all three models are denoted in black, grey parts are shared between two variants. Parts being unique for one variant are either depicted in blue (A), in green (B), or red (C).

model due to syntactic as well as semantic *merge conflicts*.

Merge Conflicts. Unify-merging of models $m_1, m_2, \ldots m_N$ using matching M yields a model $m_{\langle 1..N\rangle}$ which contains, for each tuple T in M, a model element e_T unifying all elements $e_i \in E_i$ grouped in T. Such a "copy-as-is" approach frequently fails to produce useful representations as it potentially yields syntactically ill-formed and/or semantically incorrect merged models. Hence, for a merged model to constitute a useful representation, we further require *correctness* in terms of syntactic validity (i.e., conformance to a meta model) and to semantically reflect the input models (i.e., the set of valid instantiations of the merged model is equivalent to the union of the sets of valid instantiations of all N model variants). Reconsidering our example in Fig. 2, we observe several syntactical conflicts as class diagrams do not allow classes to be both *concrete* and *abstract*, references with multiple contradicting properties, and one method declaration with multiple names. Furthermore adding the new attribute *limit* from variant *(C)* into class *Account* of the merged model leads a *semantic conflict* as new *new* instantiations can be derived from the merged model, not being valid for any input model variant. Hence, for ensuring correctness, additional steps may be required for resolving *merge conflicts*, by re-separating syntactically or semantically incompatible elements in $m_{\langle 1..N\rangle}$, although being grouped in a given matching M. For instance, in the merged model $m_{im\langle A,B,C\rangle}$ in Fig. 2 on the bottom, conflicting elements have been duplicated to obtain an—at least syntactically—correct merged model. As a consequence, merge precision is decreased to $\frac{|M|}{|E_{m_{im}}|} = \frac{13}{20} = 0.65$ as similarity information of matching M is not entirely preserved in the merged model. In addition, semantic conflicts often remain unresolved by most default strategies using simple duplication: in $m_{im\langle A,B,C\rangle}$ the duplication of reference *owner* from class *Account* within the (merged) class *Client* results in invalid instantiations of class *Account*.

To summarize, not only scalability vs. precision, but also correctness vs. precision are, in general, contradicting goals of N-way model-merging. To tackle these challenges,

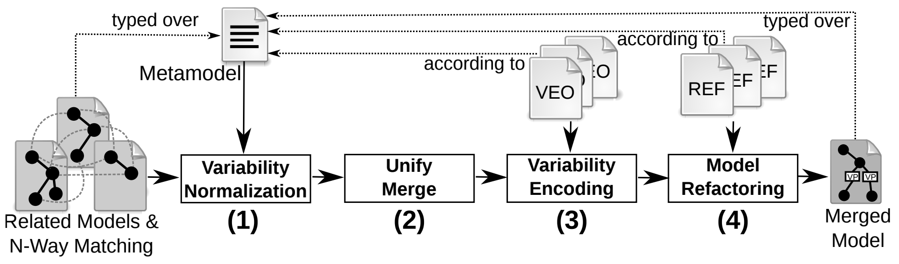


Figure 3 – Conceptual Overview of our Methodology

we propose an alternative approach for incrementally constructing correct and precise N-way merged models from any given—thus arbitrarily (im)precise— matching M in a scalable and automated way.

Improving Merge Precision by Model Refactoring. In contrast to all recent approaches, we focus on the merge-step by incrementally applying as a post-processing step model-refactoring operators, as being available for most mature modeling languages, to identify and unify further similarities among (initially) unmatched model elements. Many model-refactoring operators for UML class diagrams aim at factoring out—and thereby reusing—similar elements among structurally related model parts by employing built-in generalization constructs such as inheritance [SPLTJ01]. However, for those operators to be able to automatically detect and apply refactorings within merged models, those models must be (at least syntactically) correct. For instance, we may apply the following refactoring operations to both *Client* classes in $m_{im\langle A,B,C\rangle}$ in Fig. 2 (on the bottom): *extractSuperClass*(Client,Client,NewSuperCl), *pullUpAttribute*(name, Client, Client, NewSuperCl), *pullUpMethod*(addAccount, Client, Client, NewSuperCl) and *pullUpAssociation*(accounts, Client, Client, NewSuperCl) to identify and factor out similar elements, thus leading to an improved merge precision of $\frac{|M|}{|E_{m_{im}}|} = \frac{13}{20-3} = 0.76$ as compared to 0.65. In contrast, associations *clients* and *owner* cannot be merged in this way as those elements are duplicated within the classes *Account* and *Bank*. Hence, for model-refactoring operators to be applicable to a unify-merged model and to produce correct results even in the presence of merge conflicts, we will utilize the concept of *variability encoding* as known from product-line engineering (see Sect. 3). This enables us to also pull up both associations *clients* and *owner* to respective common superclasses of *Bank* and *Account*, thus further improving merge precision to $\frac{|M|}{|E_{m_{im}}|} = \frac{13}{17-2} = 0.87$. In contrast, some syntactical merge conflicts like contradicting multiplicities of reference *owner* and differing names of method *withdraw* cannot be handled this way thus leading to a decreased merge precision. Nevertheless, as our evaluation results will show (see Sect. 4), this is only the case for (near) optimal matchings which are usually not available in practice.

3 Improving Precision of N-Way Model Merging

Figure 3 provides a conceptual overview of our methodology. The input consists of a set of N *related models* $m_1, m_2, \ldots m_N$ which are typed over a common *meta model MM* and a *matching M* for these models. The preceding *compare*- and *match*-steps are not shown in Fig. 3 as our methodology works with any given matching (including the most imprecise one containing only singleton tuples). The merged output model $m_{\langle 1..N\rangle}$, also typed over *MM*, is constructed in two phases, where *variability encoding* (steps (1), (2) and (3) in Fig. 3) is concerned with *correctness* and *model refactoring* (step (4)) is concerned with *precision* of the merged model. The goals of the four steps are as follows.

1. **Variability Normalization.** The Precision of matching M is reduced to a *normalized matching* M' to be conforming with MM.
2. **Unify Merge.** A unify-merged model $m_{u\langle 1..N\rangle}$ is derived from the normalized matching M' as usual.
3. **Variability Encoding.** The unify-merged model $m_{u\langle 1..N\rangle}$ is enriched by encoding variant/version information to obtain a semantically well-formed merged model $m_{e\langle 1..N\rangle}$.
4. **Model Refactoring.** The variability-encoded merged model $m_{e\langle 1..N\rangle}$ is incrementally refactored to improve precision while preserving correctness to finally obtain the merged output model $m_{r\langle 1..N\rangle}$.

The following sections describe each step in greater detail.

3.1 Variability Normalization

The goal of step (1) is to ensure that for a given N-way matching the resulting merged model is *syntactically well-formed* with respect to MM after step (2). This step consists of a catalog of model-transformation rules for *normalizing* matching M by removing all tuples T from M which are not directly unifiable into a unify-merged model in compliance with MM. All such tuples are replaced by $|T|$ singleton tuples, each containing one of the matched elements in $e \in T$.

Decomposition. We assume EMOF-based representations of models by means of abstract syntax graphs constructed by atomic operations [KKT13]. We decompose input models according to the (generic) `element` types defined by MM following the concept of *atomic model elements* (*AME*) [MZB+15]. Each AME is either of type *Class* (not to be confused with UML classes), *Attribute* or *Reference*. Hence, each node of the abstract syntax graph is represented by a (meta-)*class* object, each edge by a *Reference* object, each attribute by an *Attribute* object and each element (except for the root element) references its *container* element.

Normalization Rules. We define three generic normalization rules, each rule splits up tuples in the matching by replacing them with respective singleton tuples if the grouped elements are not unifiable into a syntactically valid merged model. In particular, tuple T is split up in matching M due to possible merge conflicts originating from (another) tuple T', if

- the container-elements of all elements in T are not all matched in the same tuple $T' \in M$, or
- the attributes of all container-elements in T are not all matched in the same tuple $T' \in M$ (due to different attribute values) or the number of different values exceeds the upper-bound limit of the attribute type, or
- the references to all source-elements in T are not all matched in the same tuple $T' \in M$ (due to different targets) or the number of different targets exceeds the upper-bound limit of the reference type.

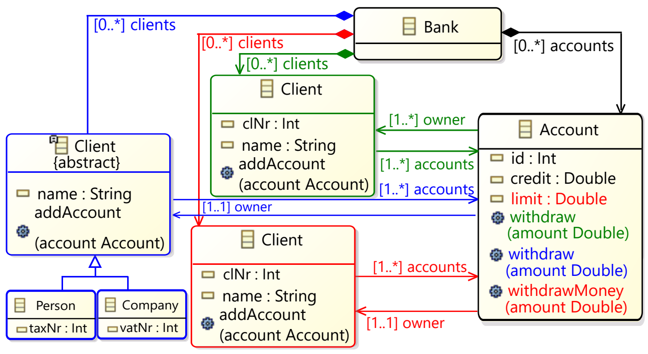


Figure 4 – Unified Model m_u after Steps (1) and (2)

These rules are recursively applied as long as some rule remains applicable.

Given the matching M in Fig. 2 (upper), the following normalization-rule applications are executed, resulting in a normalized matching M'. First, the tuple containing association *owner* from class Account to class Client is split up due to conflicting upper bounds (1 vs. $*$). Similarly, tuples containing classes Client are split up in case of conflicting values for attribute abstract (true for variant (A) and false, otherwise). Tuples containing elements related to class *Client* are thus also split up including each contained element of *Client* (i.e., attributes *clnr* and *name*, operation *addAccount* as well as association *accounts* to class *Account*). Further tuples containing references to *Client* are then split up as well (i.e., association *clients* from *Bank* to *Client*). Finally, the tuple containing the methods withdraw and withdrawMoney is split up due to conflicting names.

3.2 Unify Merge

Each tuple $T \in M'$ in a normalized matching M' is inserted as element e_T into a unify-merged model. In addition, as preparation for step (3), each element e_i in every tuple $T \in M'$ is tagged with *variability information*, uniquely identifying the model variant(s) m_i the elements originate from. Technically, we may utilize meta-information capabilities as provided by any mature EMOF-based modeling language (e.g., instances of (meta-)class *Comment* in case of UML class diagrams). In the unify-merged model of our example in Fig. 4, *variability information* is depicted with the same colors as in Fig. 2. In product-line engineering, such variability information is usually represented as *presence condition* (e.g., propositional formulae over variability parameters, e.g., Boolean features [CA05]). Although the merged model resulting from step (1) and (2) is *syntactically valid*, it may be still *semantically incorrect*, e.g., permitting model instantiations that are not valid for any model variant. For instance, the merged model in Fig. 4 allows an instance of class *Person*, which is only valid for variant *(A)*, to reference an instance of *Account* incorporating the *limit* attribute which is, however, only valid for variant *(C)*. A solution to this problem is the goal of step (3).

3.3 Variability Encoding

In this step, we employ principles from product-line engineering, called *variability encoding* [PS08]. The idea of variability encoding is to integrate variability (meta-

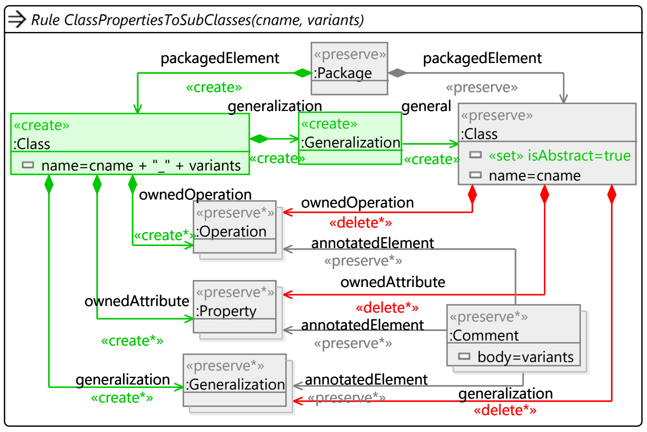


Figure 5 – VEO for UML Class Properties

)information into variable parts of programs or models by using built-in constructs of the programming- or modeling-language under consideration. One major advantage of this approach is that existing state-of-the-art tools and techniques for the development and analysis of programs/models are likewise applicable to variability-enriched model variants/versions (e.g., facilitating family-based analysis of entire product families in a single run [TAKS14]). A disadvantage arises from the additional overhead introduced by the encoded variability information. For instance, compile-time variability by means of `ifdef` macros for conditional compilation as provided by the C preprocessor can be encoded by regular `if` statements over variability parameters, thus yielding run-time variability [vRTS$^+$16]. Similarly techniques exist for encoding variability into others kinds of programs and models, by exploiting respective language constructs for supporting structured encapsulation and reuse of common and variable parts [EW11]. In particular, UML offers a variety of modeling constructs for expressing common or variable parts or behaviors (e.g., type hierarchies and inheritance in UML class diagrams and guarded transitions as well as choice-connectors in UML state machines) [NSC$^+$12, RF11]. Based on those constructs, we consider language-specific *variability-encoding operators* (*VEOs*) to encode variability meta-information of step (2) into the merged model.

VEOs. Each VEO defines a *pattern* where the operator can be applied in a merged model and *transformations* to encode variability information.

Specifying appropriate (i.e., correct and complete) catalogs of VEOs for a modeling language is subject to work on product-line engineering and, therefore, not further discussed in this paper [EW11]. Conversely, each kind of element for which no VEO can be defined has to be handled by normalization already during step (1). Hence, the more elements can be handled by encoding variability in the merged model, the less similarity information from the matching is lost in the final merged model. For instance, if no VEO is defined for similar, yet differently named methods of matched classes, the tuple grouping the classes *Account* has to be split up in step (1) due to conflicting method names withdraw and withDrawMoney thus causing further tuples to be split up (e.g., reference *accounts* from *Bank* to *Account*). Hence, if no VEO are given at all, all tuples are split up into singletons thus leading to the most imprecise merged model. We, again, consider VEOs to be implemented as graph transformation rules, allowing a fine-grained and declarative way of specifying change operations,

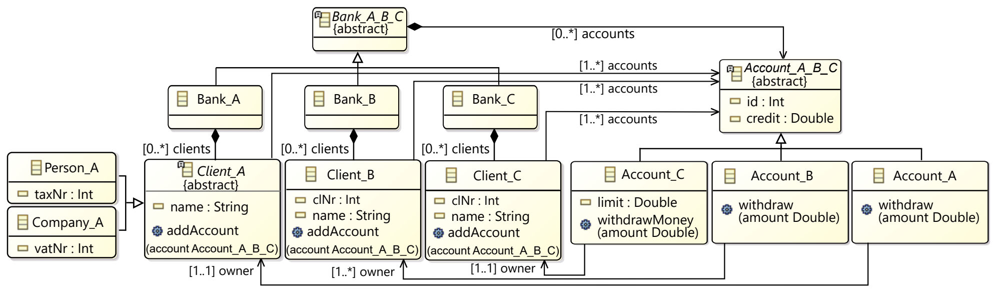


Figure 6 – Variability-Encoded Unify-Merged Model m_e after Step (3)

including (complex) application conditions and (optional) amalgamation concepts [HHT96].

VEOs for UML Class Diagrams. We illustrate VEOs for UML class diagrams as used in our tool and experiments (see Sect. 4). Our VEO catalog for class diagrams employs *inheritance* for encoding variability among elements at class level. We present two exemplary VEOs. The complete catalog as used for the subject systems in our experiments is available on-line [acc19].

- **PushPropertiesToSubClasses.** This operator (cf. Fig. 5) encodes variable properties of classes. The rule searches for all class members annotated with the same variability information, i.e., which originate from the same sets of variants. For encoding the variability information, a fresh class is created, whose name contains the original class name combined with the respective names of the variants. This new class becomes a direct sublcass of the original class which is now declared *abstract* and all class properties shared among the involved variants are pulled up into the new class.

- **RenameNamedElement**. This operator performs renaming of *NamedElements* to encode variant information following an appropriate naming convention. This is used, e.g., for *Enumerations* and *Packages* not being part of all variants to ensure uniqueness of name-space declarations.

Rule applications of the complete catalog are orchestrated such that changes are only performed if necessary (e.g., rule *PushPropertiesToSubClasses* only creates subclasses if this has not been done by previous rule applications). Applying our VEOs to the unify-merged model m_u in Fig. 4 results in the variability-encoded unify-merged model m_e in Fig. 6. Classes *Bank*, *Client* and *Account* are duplicated for each variant, where the elements shared by all variants are factored out into a common super-class (e.g., reference *accounts* from *Bank* to *Account* as well as the attributes *id* and *credit*). Although precision $\frac{|M|}{|e_{m_e}|} = \frac{13}{26} = 0.5$ of m_e remains similar to m_u, it incorporates additional overhead for the encoding (i.e., 14 classes instead of 7). Hence, step (1) may decrease matching- (and therefore merge-) precision, whereas steps (2) and (3) may increase sizes of merged models in (potentially) unnecessary ways. For instance, for our example in Fig. 6 we have $\frac{|M|}{|e_{m_e}|} = \frac{13}{26} = 0.5$. The concluding step (4) is concerned with improving precision and reducing overhead by further transforming the variability-encoded unify-merged model in a correctness-preserving manner, i.e., by factorizing further similarities among variant-specific model parts.

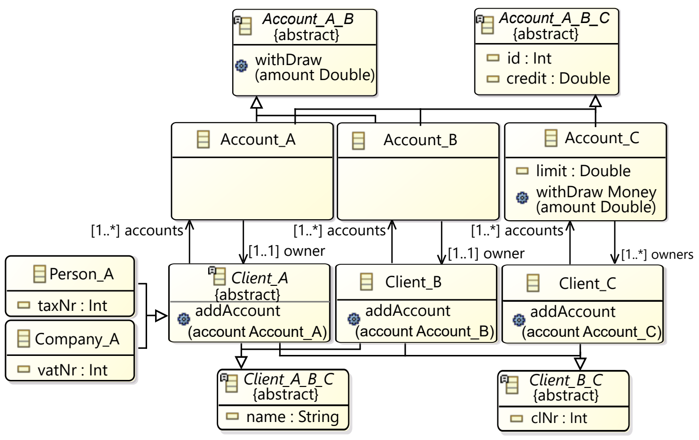


Figure 7 – Merged Model m_r after Refactoring (Excerpt)

3.4 Model Refactoring

The previous step allows us to employ default model-refactoring operators (REF), which have been extensively studied for various domains and modeling languages [SPLTJ01, MB05, SBRCT08a, ARK$^+$16]. Model-refactoring operators perform semantic-preserving model transformations, where the goal of most exiting operators is either (1) to improve models w.r.t. structural quality criteria (e.g., coupling/cohesion) and/or (2) to identify and eliminate duplicated model parts by employing language-specific reuse constructs. We consider the latter category in step (4) for enhancing merge precision. Similarity information among model variants as initially provided by groupings of the matching, but potentially being lost in steps (1) to (3), may be, at least up to a feasible degree, restored by those operators. In this regard, step (4) can be seen as a-posteriori compare/match-step to not only preserve, but even improve precision of initially imprecise or even non-available matchings. However, this novel approach is only possible in combination with variability information added in steps (2) and (3).

REF for UML Class Diagrams. We utilize well-known refactoring operators [Fow99, SPLTJ01] for class diagrams, again, defined as graph-transformation rules.

- **ExtractSuperClass.** If two classes share at least one equal property, then a fresh common super-class is introduced.
- **PullUpProperty.** If all subclasses of a common super-class share equal properties, then these properties are moved into the super-class.
- **RemoveEmptyClass.** If a class has no properties and incoming relations, then it is deleted from the model.

Similarly to VEOs, each kind of REF is defined for each type of UML model element (e.g., attributes and methods, see [acc19]).

Again, orchestration of rule applications is required as refactorings may only become applicable after other ones have been successfully applied (e.g., *RemoveEmptyClass* usually requires multiple preceding applications of *PullUpProperty*). More specifically, only those empty classes are removed which have been introduced by previous refactorings. For step (4) to yield (presumably) optimally precise results, refactoring rules are recursively applied as long as at least one rule is applicable.

An excerpt of the resulting (refactored) merged model m_r, consisting of all changed classes, is depicted in Fig 7. First, attribute *name* of all class variants of *Client* can be pulled up into a (new) common superclass *Client_ A_ B_ C* and the same for method *addAccount()*. Additionally, association *clients* from all class variants of *Bank* to respective variants of *Client* are extracted to now connect *Bank_ A_ B_ C* and *Client_ A_ B_ C* and, similarly for associations *account* now connecting *Client_ A_ B_ C* and *Account_ A_ B_ C*. Finally, association *owner*, attribute *clnr* and method *withdraw()* can been extracted into new superclasses. As a result, precision of the refactored model m_r is enhanced from $\frac{|M|}{|p_{m_e}|} = 0.65$ to $\frac{|M|}{|p_{m_r}|} = \frac{13}{26-11} = 0.87$.

4 Experimental Evaluation

For evaluating our methodology, we investigate the following aspects.

Effectiveness. The main goal of N-way model merging is to match and merge as many similar elements among the given input models as possible. Hence, we measure the effectiveness of our approach in terms of the ability to preserve (or even improve) precision of a matching in the resulting merged model.

Efficiency. To evaluate efficiency, we measure the computational effort as well as potential overhead introduced by variability encoding and model refactoring.

4.1 Oracles

Although the *compare*-operator effectively relies on similarity metrics, there exist no precise and generally accepted operational definition of an *optimal* matching in the recent literature. Instead, in the context of (automated) matching approaches and algorithms, *oracles* are frequently utilized for evaluating quality of matching using information retrieval measures [MKB09, KRG+13]. To this end, we may define matching precision as usual by

$$Prec(M) = \frac{TP}{TP + FP}$$

where *TP* denotes the number of *true positives* (matched elements deemed as similar) and *FP* denotes the number of *false positives* (matched elements deemed as dissimilar).

For each of our subject systems, we differentiate whether the used oracle is considered *optimal* due to identity-based matchings in case of generated/extracted artefacts [LBL+14, VHLFF14, MAZ17] or *approximate* due to being manually created by the developers [CCGI11] (see Table 1). This allows us to reason about the impact of the precision quality of given matching on the merge precision.

4.2 Subject Systems

We consider three subject systems, each comprising variants of class-diagrams of real-world systems from academia and industry (see Table 1). We selected the systems due to their significantly differing modeling practices and creation processes.

Pick-and-Place Unit (PPU). The *PPU* represents the domain of industrial production and automation software systems [VHLFF14, LBL+14]. The PPU handles work pieces consisting of different materials. Due to variable requirements, the PPU incorporates 13 variants and consists of 418 classes and 969 class members overall.

Barbados Car Crash Crisis Management System (bCMS). The *bCMS* describes operations of a police and fire department in crisis situations [CCGI11], where we consider class diagrams denoting the domain and architecture. Due to variable combinations of several crisis situations and parties involved, the *bCMS* consists of 15 variants with respective elements. In contrast to the *PPU* system, methods constitute the most relevant element type (see last column).

ArgoUML. ArgoUML is an UML modeling tool including all standard UML 1.4 diagrams. We use the reverse-engineered UML class-diagram representation of the Java implementation [MAZ17]. Here, 7 variants have been extracted by removing specific features for supporting different UML diagrams (i.e., activity or sequence diagrams). Due to the enormous size of its Java implementation, the variants contain more than 10000 classes and 100000 elements.

4.3 Research Questions and Methodology

We consider the following research questions in our experiments.

			#Elements			
System	Oracle	#Models	Class	Attr	Assoc	Method
PPU	Optimal	13	418	309	428	232
BCMS	Approx.	15	948	635	184	1675
ArgoUML	Optimal	7	12270	22172	4851	89566

Table 1 – Study Subjects Properties.

RQ1 (Precision). How precise is the merged model resulting from our methodology, as compared to arbitrarily (im-)precise matchings?
RQ2 (Computational Effort). Does our approach scale to large input models for arbitrarily (im-)precise matchings?
RQ3 (Matching Impact). How does varying precision of the matching impact computational effort and precision of the merged model?

Methodology. For each type of element (e.g., class members) occurring in our subject systems, we implemented respective VEO as well as REF as Henshin [ABJ$^+$10] graph-transformation rules (see Sect. 3). To obtain meaningful results, we applied the REF catalog to each input model prior to merging. The matching precision is measured using respective oracles for each subject system as described earlier. We measure *precision* of a merged model with respect to a given matching as described in Sect. 2, by comparing the number of tuples in the matching with the resulting number of (unified) elements in the merged model. To this end, we focus on class members (e.g., attributes, associations and methods) as depicted in Tab. 1. Furthermore, we consider the number of classes of our merged models to measure the potential *overhead* [EMCLGP09] introduced by variability encoding and refactoring, as compared to the unify-merged model. Regarding *computational effort*, we consider the overall CPU time as well as CPU time for each step in Fig. 3. For *RQ1*, we use three different matching approaches: identity-based (*ID*), EMFCompare (*EMFC*) [1] and similarity-based matching (*SIM*) [KKPS12]). For *RQ2*, we investigate scalability to real-world sized models by considering PPU and bCMS. Furthermore, due to its reverse-engineered nature and enormous size, we use ArgoUML to investigate limitations of our approach. Regarding *RQ3*, we (randomly) split up tuples in our matchings into singletons to artificially decrease matching precision, where we either preserve up to 50% of tuples as compared to the

[1] https://www.eclipse.org/emf/compare/

	Matching		# Class Members			Merging Precision	
System	Approach	Precision	MAT	UNI	REF	UNI	REF
PPU	ID	**1.00**	69	125	79	0.55	0.87
	SIM	0.67	93	323	79	0.21	**0.87**
	EMFC	0.15	451	451	79	0.15	**0.87**
	OR50	0.48	171	171	91	0.40	**0.76**
	Empty	0.00	686	686	662	0.10	**0.11**
bCMS	ID	1.00	604	2018	339	0.30	**1.78**
	SIM	0.51	1023	1245	332	0.48	**1.81**
	EMFC	0.29	428	2058	341	0.29	**1.77**
	OR50	0.39	2019	2019	339	0.29	**1.78**
	Empty	0.00	2310	2310	2310	0.26	**0.26**
ArgoUML	ID	**1.00**	16917	16952	16301	0.98	0.99

Table 2 – Effectiveness Results (RQ1 / RQ3).

oracle (*OR50*) or split up all tuples (*Empty*). We do not split up tuples which are needed, e.g. the container package(s) as well as data types, and subsequently split up tuples during normalization as described in Sect. 3, which may result in a matching precision less than 50 percent. All experiments have been executed on a Linux 4.18 machine with Intel Core i7-6700K and 16GB of RAM.

4.4 Results and Discussion

The results are summarized in Table 2 and Table 3. The first three columns denote input parameters as well as matching precision, the following columns depict the number of elements (e.g., class members or classes) of matchings (**MAT**) and merged models (**UNI**fied, **ENC**oded, **REF**actored). The remaining columns denote merging precision or CPU time of each step in Fig. 3 as well as overall time.

RQ1. Our evaluation results show that the normalization step for resolving syntactic merge conflicts reduces the precision of matchings, thus resulting in more class members as compared to the matching (see column *UNI* in Table 2). However, the subsequent refactoring increases the precision, again, to a similar or even better precision in the merged model in all cases (see bold numbers in column *REF*). Moreover, even in case of matchings with low precision (e.g., EMFC in case of PPU), the refactored merged model contains the same or a similar number of class members (79 for PPU, 339/332/341 for bCMS) compared to the best available matching (oracle), thus leading to a remarkable precision improvement (up to a factor of 1.81 in case of bCMS). To summarize, the precision of matchings is either preserved in case of (usually unavailable) optimal matchings or improved by our methodology in case of realistic matchings (bold numbers in Table 2).

RQ2. We observe overall CPU times ranging from 11 seconds to 17 minutes (for PPU). In particular, most of the CPU time is consumed by encoding and refactoring (more than 90 percent on average), whereas the effort for normalization and unify-merge is not relevant in most cases. More specifically, although steps (1) to (3) are finished in reasonable time for ArgoUML, refactoring took more than 3 hours even for optimally precise identity-based matchings, due to many possible refactoring applications regarding methods. CPU time naturally increases in case of matchings with (artificially) decreased precision as they impose additional effort throughout all steps. To sum up, our approach scales to input models of realistic sizes, although to a lesser extent in case of imprecise matchings or enormous class diagrams.

System	Approach	# Classes UNI	ENC	REF	Time Consumption for each Step (s) (1)	(2)	(3)	(4)	Sum
PPU	ID	70	194	263	2.9	0.3	4.2	3.5	**11**
	SIM	55	219	476	4.3	0.3	12.9	62.4	**80**
	EMFC	37	205	579	4.3	0.3	29.9	158.0	**192**
	OR50	70	202	305	0.4	0.4	6.4	7.8	**15**
	Empty	418	418	439	2.9	0.4	257.9	795.8	**1057**
bCMS	ID	620	1085	2902	12.6	0.6	472.2	11.0	**496**
	SIM	436	916	1946	9.8	0.4	223.2	7.1	**241**
	EMFC	633	1083	2940	13.8	0.6	461.6	10.7	**487**
	OR50	685	1085	2902	2.5	0.6	470.8	11.0	**485**
	Empty	948	1109	1089	5.2	0.6	718.0	19.0	**743**
ArgoUML	ID	1782	1836	2732	123.8	23.5	2221.5	11916.1	**14285**

Table 3 – Efficiency Results (RQ2 / RQ3).

RQ3. The output merged models are affected by matchings in two ways. First, merging *precision* is obstructed by significantly (i.e., artificially) imprecise matchings (see Table 2 *Empty*), as compared to realistic matchings (see *ID*, *SIM* and *EMFC*). Second, the additional *overhead* due to variability encoding and refactoring strongly correlates with precision of the matching (see Table 3), especially in case of realistic matchings (e.g., *EMFC* or *SIM*). For example, although the refactored merged model of *PPU* incorporates the same number of class members for all three realistic matchings (see Table 2), the number of classes (see column *REF* in Table 3) differs significantly. Computational effort also highly depends on the precision of the matching which especially includes the number of the usually very costly model-transformation rule applications performed in steps (3) and (4). In addition, in case of empty matchings, step (4) has the obligation to actually mimic the missing a-priori N-way matching procedure in an a-posteriori manner, thus naturally causing high computational effort. To conclude, our evaluation results show that for realistic matchings (e.g., provided by EMFC) and for reasonable artificially created imprecise matchings (*OR50*), our methodology a) produces equally precise merged models at the expense of additional overhead and b) scales to families of real-world input models. However, we may conclude that our tool is aiming at batch scenarios rather than online usage.

4.5 Threats to Validity and Limitations

Our selection of subject systems may threaten validity and generalizability of our results. Although we limit ourselves to three application domains and creation processes, we expect similar results for other domains and modeling languages as all steps of our approach are generic and/or easily adaptable. To this end, we plan to investigate the adaptions required for applying our approach to other modeling languages and domains (e.g., FODA feature diagrams [SBRCT08b] or BPMN [WR08]). Furthermore, our catalog of VEO and REF operators also constitutes a possible threat to validity, as the catalog may (1) be over-fitted to our subject systems and/or (2) contain error-prone operators. Concerning (1), further rules can be easily integrated into our approach on demand. Concerning (2), syntactical correctness has been intensively validated during the implementation of graph transformation rules in the Henshin [ABJ$^+$10] framework, whereas for semantical correctness, we have to rely on the respective domain experts to specify correct VEOs. In contrast, we can rely on existing and well-elaborated REFs. Although the design and application of refactorings may not be straightforward in all environments and domains [KS08], further pre- and postconditions as well as different possible refactoring solutions are beyond the scope of this paper. So far, we solely investigated class members as well as the number of classes in our resulting merged model for reasoning about the precision and overhead introduced by our approach. As a future work, we plan to additionally consider a) structural quality metrics (e.g., coupling and cohesion) [GPC05] as well as b) (human) comprehensibility of the resulting merged models [EMCLGP09]. In addition, we are interested in comparing the potential overhead caused by our approach for improving imprecise matches to the effort required for the a-priori computation of optimally precise matches [RKBL19]. Although our approach can handle arbitrarily (imprecise) matchings, we have to assume input models to be at least syntactically and semantically *correct*. Finally, our catalog of normalization- and encoding-rules can be adapted for supporting OCL constraints which are currently not yet supported.

5 Related Work

Model Merging in Version Management. In version management and collaborative software development, 3-way merging is an established standard [Men02]. However, the application scenarios of 3-way merging are usually different from ours as it mostly deals with inconsistent models and/or models without well-defined semantics, thus usually working on purely syntactical level. Merge conflicts are generally interpreted as the result of flawed work-flows (e.g. unsynchronized concurrent changes), which are therefore resolved interactively [WLS$^+$12] or automatically [DRV$^+$16]. The merging of models imposes further challenges w.r.t. syntactic correctness and for displaying conflicts using concrete syntax [ASW09, Wes14]. Thus 3-way merging is not suitable for our use case of model integration. In contrast to our approach, imprecise matchings generally lead to equally imprecise and/or conflicting merges which have to be resolved manually.

N-way Model Merging. Several approaches have been proposed in recent past for constructing merged models from N existing variants. A generic problem statement is presented in [RC13a] and instantiated for UML class diagrams and state machines. Similarly, Martinez et al. present a generic framework for constructing (or extracting) merged models from a set of model variants [MZB$^+$15]. The authors introduce several operators which can be implemented in a domain-specific may, similar to *compare*, *match* and *merge* operators [RC13a]. Both approaches construct merged models from which all variants are derivable again by removing or adding those parts which are (not) present in the respective variant. However, these approaches do not focus on constructing syntactically well-formed and semantically correct merged models as done by using variability encoding in our approach. Holthusen et al. [HWL$^+$14] reverse-engineer variability in families of block diagrams based on syntactical data-flow structures, whereas Wille et al. [WTS$^+$16] consider a model-based representation of object-oriented source code. In [SRA$^+$16], similarity among structural sub-patterns in families of graph-transformation rules are exploited and matched/merged for rule variants [SRCT15]. In addition, variability information is preserved similar to our unified model after step (2), but the nature of the models considered is different to our generic approach as this work is focussed on graph-transformation rules. Ryssel et al. [RPK12] extract commonalities in MATLAB libraries and reuse them by integrating the libraries into the original variants. Finally, in [NSC$^+$12], state-charts are merged using UML variability constructs, e.g. guarded transitions, which is similar to our variability encoding step. In contrast, their merge step does not compensate for imprecise matchings, thus potentially resulting in imprecise merged models.

To summarize, all existing approaches are not directly adaptable to precise N-way Model merging, regardless of the precision of the matching as in our approach.

6 Conclusion

We proposed a novel N-way model-merging methodology for EMF-compliant models by combining variability encoding and model refactoring. Our approach receives as a set of model variants to be integrated into one merged output model together with a matching. We preserve or even improve precision of arbitrary (im)precise matchings, and, at the same time, ensure syntactic well-formedness as well as semantic correctness of the merged model. As a future work, we plan to investigate potentials for generalizing our approach by developing additional catalogs of VEO and REF operators for other modeling languages and domains (e.g. UML state machines and FODA feature models). Regarding improvements of the variability-normalization step, we plan to further split up tuples with respect to the properties of their shared elements, instead of simply proceeding with (potentially unnecessarily imprecise) singleton tuples as done in our current work. Furthermore, we are interested in reducing the computational effort by explicitly controlling applications of REF operations. To this end, we plan to further exploit the (encoded) variability information (e.g., for prioritizing refactoring of properties shared by many variants). Finally, we further plan to investigate the impact of our approach to structural model-quality metrics as well as on the comprehensibility of the resulting merged model.

Acknowledgments This work was supported by the DFG (German Research Foundation) within the CoMoVa project (grant nr 330452222). This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP 1593: Design For Future – Managed Software Evolution. This work was funded by the Hessian LOEWE initiative within the Software Factory 4.0 project.

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *MODELS*, pages 121–135. Springer, 2010. `doi:10.1007/978-3-642-16145-2_9`.

[acc19] Accompanying materials for this paper. `http://pi.informatik.uni-siegen.de/projects/variance/ecmfa19`, 2019.

[ARK+16] Bader Alkhazi, Terry Ruas, Marouane Kessentini, Manuel Wimmer, and William I. Grosky. Automated Refactoring of ATL Model Transformations: A Search-based Approach. MODELS '16, pages 295–304. ACM, 2016. URL: `http://doi.acm.org/10.1145/2976767.2976782`, `doi:10.1145/2976767.2976782`.

[ASW09] K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

[BEK+07] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. EMF model refactoring based on graph transformation concepts. *Electronic Communications of the EASST*, 3, 2007.

[BLL+13] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. Staged configuration of dynamic software product lines with complex binding time constraints. pages 1–8. ACM Press, 2013. URL: `http://dl.acm.org/citation.cfm?doid=2556624.2556627`, `doi:10.1145/2556624.2556627`.

[CA05] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE'05*, volume 3676 of *Lecture Notes in Computer Science*, pages 422 – 437. Springer-Verlag, 2005.

[CCGI11] Alfredo Capozucca, Betty Cheng, Nicolas Guelfi, and Paul Istoan. OO-SPL modelling of the focused case study. In *CMA Workshop (CMA@MODELS)*, 2011. URL: `https://orbilu.uni.lu/bitstream/10993/12572/1/bCMS-SPL-complete-submit.pdf`.

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *CSMR'13*, pages 25–34. IEEE, 2013. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6498452`.

[DRV+16] C. Debreceni, I. Ráth, D. Varró, X. De Carlos, X. Mendialdua, and S. Trujillo. *Automated Model Merge by Design Space Exploration*, pages 104–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[EMCLGP09] M. Esperanza Manso, José A. Cruz-Lemus, Marcela Genero, and Mario Piattini. Empirical Validation of Measures for UML Class Diagrams: A Meta-Analysis Study. In Michel R. V. Chaudron, editor, *Models in Software Engineering*, pages 303–313, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[EW11] Martin Erwig and Eric Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, December 2011. URL: `http://doi.acm.org/10.1145/2063239.2063245`, `doi:10.1145/2063239.2063245`.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[GPC05] Marcela Genero, Mario Piattini, and Coral Calero. A survey of metrics for uml class diagrams. *Journal of Object Technology*, 4:59–92, 2005.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundam. Inf.*, 26(3,4):287–313, December 1996. URL: `http://dl.acm.org/citation.cfm?id=2379538.2379542`.

[HWL+14] Sönke Holthusen, David Wille, Christoph Legat, Simon Beddig, Ina Schaefer, and Birgit Vogel-Heuser. Family Model Mining for Function Block Diagrams in Automation Software. SPLC '14, pages 36–43, New York, NY, USA, 2014. ACM. `doi:10.1145/2647908.2655965`.

[KKPS12] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. Adaptability of Model Comparison Tools. ASE 2012, pages 306–309, New York, NY, USA, 2012. ACM. `doi:10.1145/2351676.2351731`.

[KKT13] T. Kehrer, U. Kelter, and G. Täntzer. Consistency-preserving edit scripts in model versioning. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 191–201, November 2013.

[KRG+13] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Giuliano Antoniol, and Yann-Gael Gueheneuc. Studying software evolution of large object-oriented software systems using an ETGM algorithm. *Journal of Software: Evolution and Process*, 25(2):139–163, 2013. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.519`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.519`, `doi:10.1002/smr.519`.

[KS08] Hannes Kegel and Friedrich Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 431–440, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1368088.1368147`, `doi:10.1145/1368088.1368147`.

[LBL+14] Malte Lochau, Johannes Bürdek, Sascha Lity, Matthias Hagner, Christoph Legat, Ursula Goltz, and Andy Schürr. Applying Model-based Software Product Line Testing Approaches to the Automation Engineering Domain. *AT Journal*, (to appear), 2014.

[MAZ17] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *SPLC 2017*, pages 38–41, 2017. URL: `http://doi.acm.org/10.1145/3109729.3109748`, `doi:10.1145/3109729.3109748`.

[MB05] Slaviša Marković and Thomas Baar. Refactoring OCL Annotated UML Class Diagrams. MoDELS'05, pages 280–294, Berlin, Heidelberg, 2005. Springer-Verlag. URL: `http://dx.doi.org/10.1007/11557432_21`, `doi:10.1007/11557432_21`.

[Men02] T. Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5):449–462, 2002.

[MKB09] Thilo Mende, Rainer Koschke, and Felix Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):143–169, 2009. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.402`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.402`, `doi:10.1002/smr.402`.

[MZB+15] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. L. Traon. Automating the Extraction of Model-Based Software Product Lines from Model Variants. In *ASE 15*, pages 396–406, November 2015.

[NSC+12] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Variant Feature Specifications. *IEEE Transactions on Software Engineering*, 38(6):1355–1375, 2012.

[PS08] H. Post and C. Sinz. Configuration Lifting: Verification Meets Software Configuration. ASE '08, pages 347–350. IEEE Computer Society, 2008.

[RC13a] J. Rubin and M. Chechik. N-way Model Merging. In *Proceedings of the 2013 Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 301–311, New York, NY, USA, 2013. ACM.

[RC13b] J. Rubin and M. Chechik. Quality of Merge-Refactorings for Product Lines. In *Fundamental Approaches to Software Engineering*, number 7793 in Lecture Notes in Computer Science, pages 83–98. Springer Berlin Heidelberg, January 2013.

[RCC15] J. Rubin, K. Czarnecki, and M. Chechik. Cloned product variants: from ad-hoc to managed software product lines. *International Journal on Software Tools for Technology Transfer*, 17(5):627–646, 2015.

[RF11] Bernhard Rumpe and Robert France. Variability in UML language and semantics. *Software & Systems Modeling*, 10(4):439, Aug 2011. URL: `https://doi.org/10.1007/s10270-011-0210-3`, `doi:10.1007/s10270-011-0210-3`.

[RKBL19] Dennis Reuling, Udo Kelter, Johannes Bürdek, and Malte Lochau. Automated N-way Program Merging for Facilitating Family-based Analyses of Variant-rich Software (accepted). *ACM Trans. Softw. Eng. Methodol.*, 2019. `doi:10.1145/3313789`.

[RPK12] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic Library Migration for the Generation of Hardware-in-the-loop Models. *Sci. Comput. Program.*, 77(2):83–95, February 2012.

[SBRCT08a] Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. *Automated Merging of Feature Models Using Graph Transformations*, pages 489–505. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. URL: `https://doi.org/10.1007/978-3-540-88643-3_15`, `doi:10.1007/978-3-540-88643-3_15`.

[SBRCT08b] Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. *Automated Merging of Feature Models Using Graph Transformations*, pages 489–505. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. URL: `https://doi.org/10.1007/978-3-540-88643-3_15`, `doi:10.1007/978-3-540-88643-3_15`.

[SPLTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *International Conference on the Unified Modeling Language*, pages 134–148. Springer, 2001.

[SRA+16] D. Strüber, J. Rubin, T. Arendt, M. Chechik, F. Täntzer, and J. Plöger. RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules. In *FASE*, 2016.

[SRCT15] D. Strüber, J. Rubin, M. Chechik, and G. Täntzer. A Variability-Based Approach to Reusable and Efficient Model Transformations. In *FASE*, volume 9033 of *Lecture Notes in Computer Science*, pages 283–298. Springer Berlin Heidelberg, 2015.

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014. `doi:10.1145/2580950`.

[TAKS14] T. Thüm, S. Apel, C. Kästner, and G. Schaefer, I.and Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

[TS14] C. M T and E. Sherly. Refactoring sequence diagrams for code generation in UML models. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 208–212, Sep. 2014. `doi:10.1109/ICACCI.2014.6968414`.

[VHLFF14] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit. Technical Report TUM-AIS-TR-01-14-02, TU München, 2014.

[vRTS+16] Alexander von Rhein, Thomas Thuem, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming*, 85(1, Part 2):125 – 145, 2016. URL: `http://www.sciencedirect.com/science/article/pii/S2352220815000577`, `doi:https://doi.org/10.1016/j.jlamp.2015.06.007`.

[Wes14] B. Westfechtel. Merging of EMF models. *Software & Systems Modeling*, 13(2):757–788, 2014.

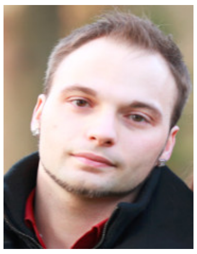
[WLS+12] K. Wieland, P. Langer, M. Seidl, M. Wimmer, and G. Kappel. Turning Conflicts into Collaboration. *Computer Supported Cooperative Work (CSCW)*, 22(2-3):181–240, September 2012.

[WR08] Barbara Weber and Manfred Reichert. Refactoring Process Models in Large Process Repositories. In Zohra Bellahsène and Michel Léonard, editors, *Advanced Information Systems Engineering*, pages 124–139, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[WTS+16] D. Wille, M. Tiede, S. Schulze, C. Seidl, and I. Schaefer. *Identifying Variability in Object-Oriented Code Using Model-Based Code Mining*, pages 547–562. Springer International Publishing, 2016.

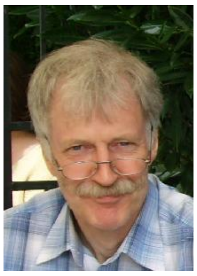
[WWS+17] David Wille, Kenny Wehling, Christoph Seidl, Martin Pluchator, and Ina Schaefer. Variability Mining of Technical Architectures. SPLC '17, pages 39–48, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3106195.3106202`, `doi:10.1145/3106195.3106202`.

About the authors



Dennis Reuling is a PhD student at the Software Engineering Group in the University of Siegen. His main research interests are model merging and variability management in model-based software variants and families. He is the lead researcher in the DFG CoMoVa project. Contact him at `dreuling@informatik.uni-siegen.de`

Malte Lochau is PD at the Real-Time Systems Laboratory of Prof Andy Schürr at the TU Darmstadt. His research interests are software product line engineering, software testing, and formal semantics. His research is part of the DFG project IMoTEP, DFG SFB 1053 MAKI and LOEWE Project SF 4.0. Contact him at `malte.lochau@es.tu-darmstadt.de`



Udo Kelter holds the chair of Software Engineering and Database Systems at the University of Siegen, Germany. His main fields of research are model-based system development and version management. Contact him at `kelter@informatik.uni-siegen.de`