RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules

Daniel Strüber[1], Julia Rubin[2], Thorsten Arendt[1],
Marsha Chechik[3], Gabriele Taentzer[1], Jennifer Plöger[1]

[1] Philipps-Universität Marburg, Germany. {strueber, arendt, taentzer, ploeger1}@informatik.uni-marburg.de
[2] Massachusetts Institute of Technology, USA. mjulia@csail.mit.edu
[3] University of Toronto, Canada. chechik@cs.toronto.edu

Abstract. Unifying similar model transformation rules into variability-based ones can improve both the maintainability and the performance of a model transformation system. Yet, manual identification and unification of such similar rules is a tedious and error-prone task. In this paper, we propose a novel merge-refactoring approach for automating this task. The approach employs *clone detection* for identifying overlapping rule portions and *clustering* for selecting groups of rules to be unified. Our instantiation of the approach harnesses state-of-the-art clone detection and clustering techniques and includes a specialized *merge construction* algorithm. We formally prove correctness of the approach and demonstrate its ability to produce high-quality outcomes in two real-life case-studies.

1 Introduction

Model transformation is a key enabling technology for Model-Driven Engineering, pervasive in all of its activities, including the translation, optimization, and synchronization of models [1]. Algebraic graph transformation (AGT) is one of the main paradigms in model transformation, allowing rules to be specified in a high-level, declarative manner [2]. Recently, many complex transformations have been implemented using AGT [3,4,5]. AGT is gaining further importance due to its use as an analysis back-end for imperative transformation languages [6].

Transformation systems often contain rules that are substantially similar to each other. Yet, until recently, various model transformation languages lacked constructs suited to capture these similar *rule variants* in a compact manner [7]. The most frequently applied mechanism for creating variants was cloning: developers produced rules by copying and modifying existing ones. The drawbacks of cloning are well-known, e.g., the need to update all clones when a bug is found in one of the variants. Furthermore, creating a large set of mutually similar rules also impairs the performance of transformation systems: each additional rule increases the computational effort, possibly rendering the entire transformation infeasible. Blouin et al. report that to be the case with as few as 250 rules [8].

Variability-based (VB) rules are an approach to address these issues [9]. Inspired by product line engineering (PLE) principles [10,11], a VB rule encodes a

set of rule variants in a single-copy representation, explicating common and variable portions. In [9], we provide an algorithm for applying VB rules and show that it outperforms the application of classical rules in terms of execution time.

The VB rules in [9] were created manually, a tedious and error-prone task relying on the precise identification of (i) sets of rule variants, each to be unified into a single VB rule; (ii) rule portions that should be merged versus portions that should remain isolated. The choices made during these steps have a substantial impact on the quality of the produced rules.

In this work, we present *RuleMerger*, a novel approach for automating the merge-refactoring of model transformation rules. The approach includes a three-component framework (see Fig. 1). It applies *clone detection* [12] to identify overlapping portions between rules and *clustering* [13] to identify disjoint groups of similar rules. During *merge construction*, common portions are unified and variable ones are annotated to create VB rules. Each component can be instantiated and customized with respect to specific quality goals, e.g., to produce rules optimized for background execution or easy editing. Since the framework guarantees that all created rule sets are semantically equivalent, we envision a system that enables users to edit rules in a convenient representation and to automatically derive a highly efficient one.

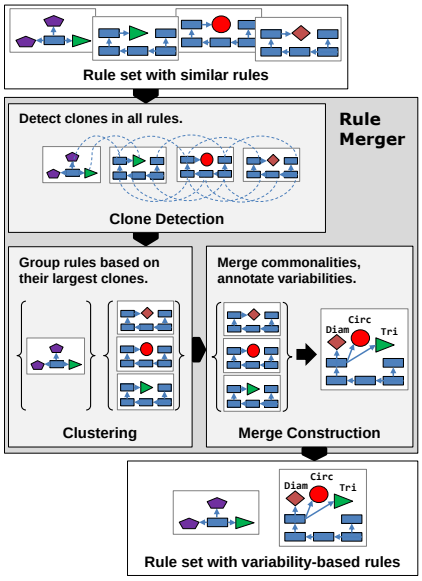

Fig. 1. Overview of *RuleMerger*

The distinguishing factors of this approach, compared to merge-refactoring approaches in the PLE domain [14,15,16], are its ability to detect overlapping *portions* rather than pairs of similar elements and to create *multiple* output VB rules rather than one single-copy representation of all rules. These factors allow us to address the performance and maintainability issues related to cloning.

Contributions. This paper makes the following contributions: (1) It presents a novel merge-refactoring approach for AGT-based model transformation rules. (2) It formally proves the correctness of the approach, showing the equivalence of the produced VB rules to their classical counterparts. (3) It instantiates the approach by providing a novel *merge construction* algorithm and harnessing state-of-the-art clone detection and clustering techniques. (4) It empirically shows that the approach allows producing VB rules being superior to their classical counterparts in terms of execution time and the amount of contained redundancy.

The rest of this paper is structured as follows: Sec. 2 introduces a running example. In Sec. 3, we fix preliminaries. In Sec. 4, we outline the approach and argue for its correctness. Sec. 5 reports on our instantiation of *RuleMerger*. Sec. 6 presents our evaluation. In Sec. 7 and 8, we discuss related work and conclude.

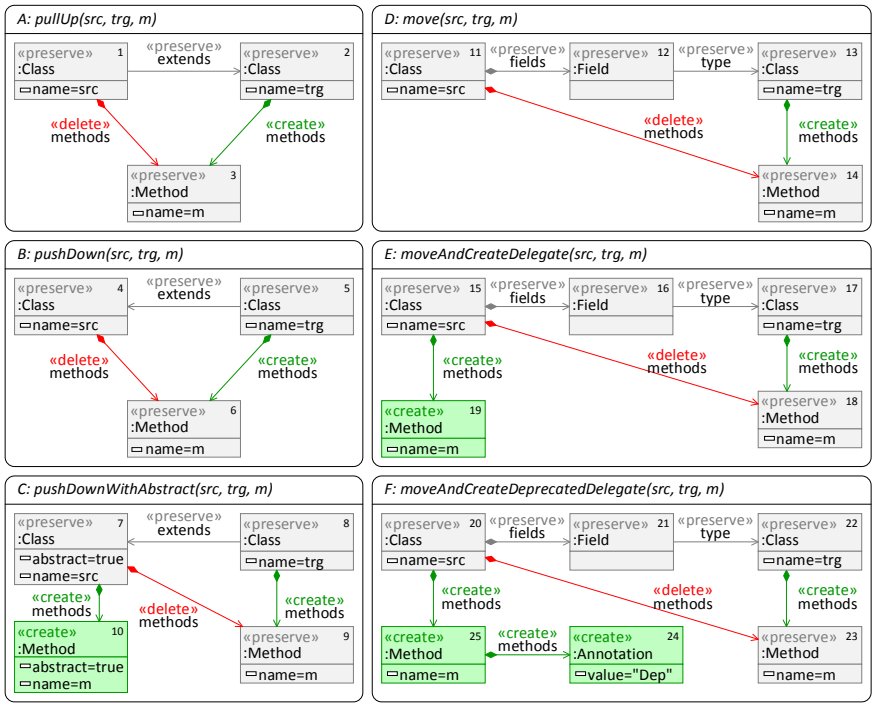

Fig. 2. Original transformation rules.

2 Running example

Consider a set of model transformation rules aimed at improving the structure of an existing code base by using *refactoring* [17]. Fig. 2 shows six refactoring rules expressed in an abstract syntax notation [2]. The rules describe several simple ways of relocating a method between different classes. We present the rules in an integrated form, with the left- and right-hand sides of the transformation being represented in one graph. The left-hand side of a rule comprises all *delete* and *preserve* objects. The right-hand side contains all *preserve* and *create* objects.

Rule A takes as input two classes, one of them sub-classing the other, and a method. Each of these input objects is specified by its name. The rule moves the method from a sub-class to its super-class, by deleting it from the sub-class and adding it to the super-class. Similarly, rule B moves a method from the super-class to one of its sub-classes. Rule C also moves a method from the super- to a sub-class, but, in addition, creates an abstract method with the same name in the super-class. Rules D, E and F move a method across an association. The latter two rules also create a "wrapper" method of the same name in the source class. Rule F uses an annotation to mark this "wrapper" method as deprecated.

Such rule sets are often created by cloning, that is, copying a seed rule and modifying it to fit the new purpose. We consider the merge-refactoring of a rule set created using cloning. The result is a rule set with variability-based (VB) rules in which the common portions are unified and the differences are explicated, as shown in Fig. 3. Specifically, rules B and C are merged, producing a new VB rule $B{+}C$. Rules D, E, and F are merged into $D{+}E{+}F$. Rule A remains as is.

Each VB rule has a set of *variation points*, corresponding to the names of the original rules: Rule $B{+}C$ has the variation points B and C. In addition, each rule has a *variability model* specifying relations between variation points, such as their mutual exclusion: $B{+}C$ has the variability model *xor(B,C)*. VB rules are *configured* by binding each variation point to either *true* or *false*. Portions of VB rules are annotated with *presence conditions*. These portions are removed if the presence condition evaluates to *false* for the given configuration. Element #32 and its incoming edge, both annotated with C, are removed in the configuration $\{C{=}false,\ B{=}true\}$. These VB rules offer several benefits w.r.t. maintainability: The amount of redundancy is reduced, ensuring consistency between variants during changes; bugs are fixed in one place. The total number of rules is smaller.

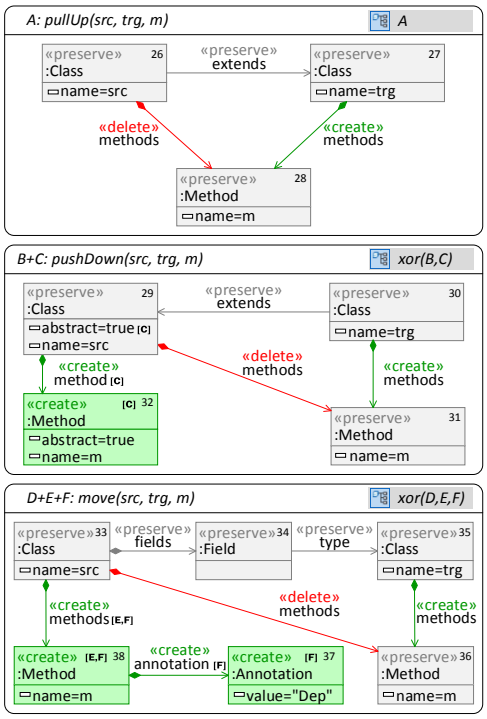

Fig. 3. Variability-based rules.

In this example, the user selects and configures one of theses rule at a time, to derive one specific rule variant – a process similar to that in PLE approaches [11]. In an alternative use-case, *all* rules of a rule set may be applied simultaneously. Configurations can then be determined automatically by the transformation engine [9], leading to considerable performance savings: The application sites or *matches* for the common portions are identified first and used as starting points for matching the variable portions. Such cases are demonstrated in Sec. 6.

3 Preliminaries: Variability-based model transformation

We now give preliminaries, starting with simple transformation rules.

Definition 1 (Rule) *A* rule $r = L \xleftarrow{le} I \xrightarrow{ri} R$ *consists of graphs* L, I *and* R, *called* left-hand side, interface graph *and* right-hand side, *respectively, and two injective graph morphisms,* le *and* ri. *A rule is* connected *iff, treating all edges as undirected,* $\forall G \in \{L, R\}$ *there is a path between each pair of nodes in* G.

The rules in Fig. 2 follow this definition. Elements of I are annotated with the action *preserve*, elements of $L \setminus le(I)$ and $R \setminus ri(I)$ with *delete* and *create*.

Given a rule, a *subrule* encapsulates a subset of its actions on a substructure. To identify actions on substructures of *one* rule, we talk about subrule embeddings. For clone detection, the subrule relation must capture common actions on common patterns in *different* rules – we then talk about *subrule morphisms*.

Definition 2 (Subrule morphism) *Given a pair of rules* $r_0 = (L_0 \xleftarrow{le_0} I_0 \xrightarrow{ri_0} R_0)$ *and* $r_1 = (L_1 \xleftarrow{le_1} I_1 \xrightarrow{ri_1} R_1)$ *with injective mappings* le_i, ri_i *for* $i \in \{0,1\}$, *a* subrule mapping $s : r_0 \to r_1$, $s = (s_L, s_I, s_R)$ *consists of injective mappings* $s_L : L_0 \to L_1$, $s_I : I_0 \to I_1$, *and* $s_R : R_0 \to R_1$ *such that in the diagram in Fig. 4* (1) *and* (2) *commute. In addition, the intersection of* $s_L(L_0)$ *and* $le_1(I_1)$ *in* L_1 *as well as the intersection of* $s_R(R_0)$ *and* $ri_1(I_1)$ *in* R_1 *is isomorphic to* I_0. *Moreover,* $L_1 - (s_L(L_0) - s_L(le_0(I_0)))$ *is a valid graph.*
Subrule mapping s *is called a* subrule embedding *if all of its morphisms* s_L, s_I, *and* s_R *are inclusions. Given two subrule embeddings* $s : r_0 \to r_1$ *and* $s' : r'_0 \to r'_1$, *we have that* $s \subseteq s'$ *if there are subrule embeddings* $t_0 : r_0 \to r'_0$ *and* $t_1 : r_1 \to r'_1$ *with* $s' \circ t_0 = t_1 \circ s$.

The conditions prefaced with "in addition" ensure that a subrule always performs the same actions on related elements as the original rule and that the larger pattern of the original rule does not prevent a subrule to be applied.

For example, in Fig. 2, B is a subrule of $B{+}C$ since B can be injectively mapped to $B{+}C$. The actions on the original and mapped elements are always the same.

We capture variability in rule sets by propositional expressions over a fixed set of independent *variation points*, calling these expressions *variability conditions*.

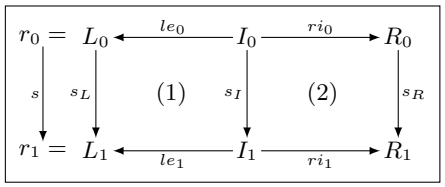

Fig. 4. Subrule morphism.

Definition 3 (Language of variability conditions) *Given a set of atomic terms* V, *called* variation points, $\mathcal{L}_V$ *is the set of all propositional expressions over* V, *called* variability conditions. *A* variability configuration *is a total function* $cfg : V \to \{true, false\}$. cfg satisfies *a variability condition* vc *if* vc *evaluates to* true *when each variation point* vp *in* vc *is substituted by* $cfg(vp)$. *A variability condition is* valid *if there is a variability configuration satisfying it. Given two variability conditions* X *and* Y, X *is* stronger *than* Y *iff* $X \implies Y$.

For example, in the rule $D{+}E{+}F$ in Fig. 2, $V = \{D, E, F\}$. *True*, E, and $E \vee F$ are valid variability conditions; $E \wedge \neg E$ is not valid. A possible configuration might bind the variation points D to *false*, E to *true* and F to *false*, which would satisfy the variability condition $E \vee F$.

In a VB rule, variability is formalized by means of subrule embeddings, each describing a single variant. The intersection of subrule embeddings is the part of the rule where all variants overlap, i.e., the *base rule*. Each subrule has a *variability condition* determining when this variant shall be active. Moreover, the entire rule has a *variability model*. The base rule does not have any annotations.

Definition 4 (Variability-based rule) *Given* $\mathcal{L}_V$, *a* VB rule $\hat{r} = (r, S, v, pc)$ *consists of a rule* r, *a set* S *of subrule embeddings to* r, *a variability condition* v, *called* variability model, *and a function* $pc : S \cup \{id_r\} \to \mathcal{L}_V$. *Function* pc *defines* presence conditions *for subrules s.t.* $pc(id_r)$ *is* true *and* $\forall s \subseteq s' : pc(s') \implies pc(s)$. *The* base rule *is determined by the intersection of all subrule embeddings.*

Rule $D{+}E{+}F$ in Fig. 3 is a compact representation of a VB rule over variation points D, E, and F with various subrule embeddings such as $\{s_E, s_{E\vee F}, s_{D\wedge E}, ...\}$. The base rule comprises all elements with the presence condition *true*: i.e., objects without annotations such as #33–36, and their relations. Elements #37 and #38 have non-*true* presence conditions and are therefore not present in all subrule embeddings. To ensure equivalence to the original three rules, the variability model v specifies mutual exclusion between variation points: $v = xor(v_D, v_E, v_F)$.

To show the correctness of our approach, we consider the *flattening* of a VB rule – an operation for generating the individual "flat" rules it represents.

Definition 5 (Flattening of a VB rule) *Let a VB rule $\hat{r} = (r, S, v, pc)$ over $\mathcal{L}_V$ be given. For each variability condition c in $\mathcal{L}_V$, the following holds: if $c \wedge v$ is valid, $S_c \subseteq S$ is a set of subrule embeddings iff $\forall s \in S : s \in S_c$ if $c \implies pc(s)$. Merging all subrule embeddings in S_c by first computing the intersections of all pairs of embeddings and merging them along these interfaces afterwards, yields a subrule embedding $r_c \to r$. r_c is the flat rule for condition c. $Flat(\hat{r})$ is the set of all flattened rules: $\{r_c \mid c \in \mathcal{L}_V \wedge\ (c \wedge v)$ is valid$\}$.*

For example, consider just the rule $D{+}E{+}F$ in Fig. 3. $c \wedge v$ becomes valid if $xor(cfg(v_D), cfg(v_E), cfg(v_F))$ is *true*. Hence, $Flat(D{+}E{+}F) = \{D, E, F\}$. In [9], it is shown that the application of a VB rule is equal to the application of flattened rules. This result is key to argue for the correctness of *RuleMerger*.

4 Framework

Given a rule set with similar rules, *RuleMerger*, outlined in Fig. 1, aims to find an efficient representation of these rules using a set of variability-based (VB) rules. At its core is a framework of three components called *clone detection*, *clustering* and *merge construction*. We specify the input and output of each component and show correctness of *RuleMerger* based on these specifications. Each component may be instantiated in various ways, as long as its specification is implemented.

4.1 Clone Detection

Clone detection allows identifying overlapping portions between the input rules. We use clone detection as a prerequisite for both clustering and merge construction: Rules with a large overlap are clustered together. Merging overlapping portions rather than individual elements allows us to preserve the essential structural information expressed in the rules. Moreover, the execution performance of the created VB rules can be considerably improved by restricting clone detection to *connected* portions: Connected patterns can be matched much more efficiently than multiple independent patterns [18].

Formally, given a set of rules, a *clone* is a largest subrule that can be embedded into a subset of this rule set. To account for the optional restriction of clone detection to connected portions, we analogously define *connected clones* based on largest connected subrules. To establish a well-defined merge construction, we define a *compatibility relation*, ensuring that two clones never assign the same object contained in one rule to diverging objects contained in another one.

Definition 6 (Clone group) *Given a set $\mathcal{R} = \{r_i | i \in I\}$ of rules, a* (connected) clone group $CG_\mathcal{R} = (r_c, \mathcal{C})$ *over $\mathcal{R}$ consists of a (connected) rule r_c, called* clone*, and set $\mathcal{C} = \{c_i | i \in I\}$ of subrule mappings $c_i : r_c \to r_i$ iff there is no set $\mathcal{C}' = \{c'_i | i \in I\}$ of subrule mappings $c'_i : r'_c \to r_i$ with a subrule mapping $i : r_c \to r'_c$ where r'_c is a (connected) rule.*

Given a clone group $CG_\mathcal{R}$ and a subset $\mathcal{R}' \subseteq \mathcal{R}$, $CG_\mathcal{R}$ is reduced *to $\mathcal{R}'$, written $Red(CG_\mathcal{R}, \mathcal{R}') = (r_c, \mathcal{C}')$, by $\mathcal{C}' = \mathcal{C} \setminus \{c_j | r_j \notin \mathcal{R}'\}$. Clone groups $CG_\mathcal{R} = (r_c, \{c_k | k \in K\})$ and $CG_{\mathcal{R}'} = (r'_c, \{c'_l | l \in L\})$ with $\mathcal{R} \subseteq \mathcal{R}$ and $K \subset L$ are* compatible *if there is a subrule mapping $in : r_c \to r'_c$ with $\forall k \in K : c_k = c'_k \circ in$.*

Table 1 shows the result of applying clone detection to rules shown in Fig. 2. Each row denotes a clone group, comprising a set of rules and a clone present in each of these rules. Clones are indicated by their *size*, calculated as the total number of involved nodes and edges. The rows are ordered by the size of the clone. In particular, CG2 represents objects #15-18, #20-23 and their interrelations. CG1 incorporates objects #19 and #25 and their incoming relationships in addition. Clone groups CG1 and CG2 are compatible: The clone of CG2 extends the one of CG1. CG2 can be reduced to rule set $\{E, F\}$ by discarding the embedding into rule D. CG2 and CG3 are not compatible: their rule sets are not in subset relation. Each clone group in Table 1 is connected.

Name	Rules	Size
CG1	{*E*, *F*}	10
CG2	{*D*, *E*, *F*}	8
CG3	{*C*, *E*, *F*}	7
CG4	{*B*, *C*}	6
CG5	{*A*, *B*, *C*, *D*, *E*, *F*}	5

Table 1. Clone groups, as reported by clone detection.

The output of clone detection is a set of clone groups – in the example, all rows of Table 1. These clone groups may be pair-wise incompatible.

4.2 Clustering

As a prerequisite for merge construction, we introduce *clustering*, an operation that splits a rule set into a cluster partition based on similarity between rules. Its input are a set of rules and a set of clone groups over these rules.

Definition 7 (Cluster) *Given a set $\mathcal{R}$ of rules and a set CG of clone groups over $\mathcal{R}$, a* cluster *Cl over $\mathcal{R}$ is a set of clone groups $CG_{\mathcal{R}'} \subset CG$ over each subset $\mathcal{R}' \subseteq \mathcal{R}$. Given a partition $\mathcal{P}$ of $\mathcal{R}$, a* cluster partition *is a set $Par(Cl)_\mathcal{P}$ of clusters over Cl where for each $P \in \mathcal{P}$ there is a cluster $Cl_P \in Par(Cl)_\mathcal{P}$ comprising clone groups $Red(CG_{\mathcal{R}'}, P)$ and $CG_{P'} \subseteq CG_P$ over subsets P' of P. Each cluster $Cl_P \in Par(Cl)_\mathcal{P}$ is called* a sub-cluster *of Cl.*

In the example, there is a cluster partition over the rule set with sub-clusters over {*A*}, {*B*, *C*}, and {*D*, *E*, *F*}. We consider the sub-cluster over {*D*, *E*, *F*}: The clone groups over this set are obtained by reducing the mappings of {CG2, CG5} to rules *D*, *E* and *F*, i.e., discarding all mappings not referring to either rule. To obtain the clone groups over subset {*E*, *F*}, we include CG1 and CG3 as well and reduce the mappings of {CG1, CG2, CG3, CG5} to *E* and *F*.

The output of clustering is one clustering partition over the rule set. Given multiple possible partitions, the instantiation of clustering has to choose one.

4.3 Merge Construction

Merge construction takes a cluster partition over the entire rule set as input. Each sub-cluster becomes a VB rule in the output. The available information on overlapping, given by clone groups, is considered to merge corresponding elements. Merging requires that the clone groups over each sub-cluster are compatible. Incompatible clone groups have to be discarded before merging, a non-trivial task requiring a strategy to determine what to discard. The instantiation in Sec. 5 provides such a strategy. To maintain traceability between original and new rules, we define a variation point for each original rule. The variability model is set over the variation points, specifying that exactly one of them is valid at a time.

Definition 8 (Cluster merge) *Given a cluster partition $Par(Cl)_{\mathcal{P}}$ over a cluster Cl over $\mathcal{R}$, each sub-cluster $Cl_P \in Par(Cl)_{\mathcal{P}}$ is merged to a variability-based rule $\hat{r} = (r, S, v, pc)$ by merging all rules in $P = \{r_j | j \in J\}$ over compatible clone groups in Cl_P. The result is a rule r. $S = \{s_i : r_i \to r\}$ consists of all resulting subrule embeddings. Variation points V are determined by the rules in P: $V = \{v_j | j \in J\}$. Moreover, $v = \mathrm{Xor}_{j \in J}(v_j)$ and $pc(s_j) = v_j$. We use the notation $Merge(Cl_P)$ to indicate $\hat{r}$ and $Merge(Cl) = \{Merge(Cl_P) | Cl_P \in Par(Cl)_{\mathcal{P}}\}$.*

Rules are merged over compatible clone groups by gluing those rule elements that are in relation via subrule mappings. This relation is extended to an equivalence relation, so in particular, the transitive closure is considered as well. All elements not in the relation are merged in disjointly.

In the example, considering all clone groups identified for the sub-cluster over {*D*, *E*, *F*}, CG1–2 are compatible; since we consider the reduction to {*D*, *E*, *F*} they are incompatible to CG3 and CG5. Merging the sub-cluster based on clone groups CG1–2 yields a VB rule isomorphic to *D*+*E*+*F* in Fig. 3. The variability model v is set to $xor(cfg(v_D), cfg(v_E), cfg(v_F))$. In the compact representation of VB rules shown in Fig. 3, the presence condition of an element is the disjunction of all variation points whose corresponding subrules contain the element.

As a key well-definedness result, we obtain that merging a rule set and then flattening it produces the original set. We provide a proof in [19].

Theorem 1 (Correctness of rule merger) *For any cluster Cl over a set $\mathcal{R}$ of flat rules, we have $Flat(Merge(Cl)) = \mathcal{R}$.*

Note that the opposite operation, first flattening a VB rule set and then merging the resulting flat rules, may not yield the same VB rule set: In general, there are several VB rules with the same flattening. In fact, Theorem 1 ensures that *all* VB rule sets created by instantiations of *RuleMerger* have the same flattening, i.e., they are semantically equivalent.

5 Instantiating RuleMerger

We now present our instantiation of the *RuleMerger* framework based on state-of-the-art *clone detection* and *clustering* algorithms and a new *merge construction* algorithm. We describe two input parameters enabling customizations with respect to specific quality goals. For implementation details, see [19].

Clone Detection. We considered the applicability of three techniques for clone detection, each of them allowing to identify *connected clones* as per Def. 6. First, we applied *gSpan*, a general-purpose graph pattern mining tool [20]. Using this tool, we experienced heap overflows even on small rule sets. Second, we re-implemented eScan [21], which terminated with *insufficient memory* errors for larger rule sets. While our implementation could be flawed, [22] reports on a similar experience with their re-implementation of eScan. Finally, we applied ConQAT [22], a heuristic technique which delivers fast performance at the expense of precision. It was able to analyze rule sets of 5000 elements in less than 10 seconds while reporting a large portion of relevant clones. We used ConQAT in our experiments on realistic rule sets.

We provide a customization to increase the speed-up produced by the constructed rules: The performance-critical task in rule application, *matching*, considers just the rule left-hand sides. Consequently, performance is optimized when rules are merged based on their overlap in left-hand sides. To this end, a Boolean parameter *restrictToLhs* allows to restrict the rule portions considered by clone detection. When set to *true*, it only finds and reports clones for left-hand sides.

Clustering. From a large variety of approaches to cluster a set of objects based on their similarity [13], we chose *AverageLinkage*, a hierarchical agglomerative method, due to its convenient application to our approach. It assumes a distance function – a measure of similarity between the clustered elements. We consider the similarity of rule pairs, defining it as the size of the rules' largest common clone divided by their average size. In the example, similarity of rules E and F is calculated based on CG1, evaluating to $\frac{10}{11} = 0.91$. It further assumes a customizable *cutting-level threshold* parameter that we describe in what follows.

The method builds a cluster hierarchy, often visualized using a *dendogram* – a tree diagram arranging the input elements, as shown in Fig. 5. Tree nodes describe proximity between rule sets. The "lower" in the tree two nodes are connected, the more similar are their corresponding rules. For example, rule D is similar to E and F, but the similarity is not as strong as that between just E and F. The clustering result is obtained by "cutting" using the cutting-level threshold, marked by a vertical bar in Fig. 5, and collecting the obtained subtrees.

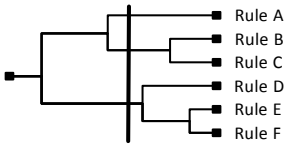

Fig. 5. Cluster dendogram, as reported by clustering.

Merge Construction. We propose a custom algorithm for merge construction. It proceeds in two steps: determining *what* is to be merged and *how* to do the merging. The first step, called *merge computation*, takes as input the cluster partition created by clustering (see Def. 7). To ensure a well-defined merge, merge computation refines the given cluster partition by discarding incompatible clone groups (Def. 6), retaining sub-clusters for which a set of compatible clone groups is available. To this end, we apply a greedy strategy that aims to capture a high degree of overlap. Each sub-cluster becomes a `MergeRule` in the output of merge computation, a `MergeSpecification`. The second step, *merge refactoring*, creates VB rules according to this `MergeSpecification` as per Def. 8.

Fig. 6 specifies a metamodel for the interface between merge computation and merge refactoring. `MergeSpecification`, corresponding to the overall rule set, acts as an overarching container for a set of `MergeRule`s. One `MergeRule` identifies a sub-cluster that is to be merged into a VB rule. In order to preserve the graphical layout of the contained rules, one rule is stated as *masterRule*; this rule is used as a starting point in creating the VB rule. To retain as much layout information as possible, it is best to select the largest input rule as the *masterRule*. A `MergeRule` specifies all elements to be unified in the created VB rule. For each element in the resulting rule, a `MergeRuleElement` is defined, referring to the elements to be represented by it. In a consistent specification, each rule element is referred to by exactly one `MergeRuleElement`.

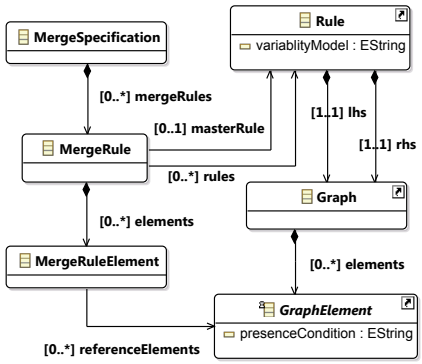

Fig. 6. MergeSpecification metamodel.

Fig. 7 sketches the merge computation algorithm. The output `MergeSpecification` is created in line 2 and incrementally filled by considering each cluster. In each iteration of the loop starting in line 5, a new sub-cluster is constructed. We apply a greedy strategy to integrate as many compatible clone groups as possible, starting with the *top* – the largest available – clone group in lines 6-8 and incrementally adding the next largest compatible ones in 9-14. For each clone group, we temporarily create a new `MergeRule`, integrating its contents with the result `MergeRule` in line 12. When no more compatible clone groups are found, we add the `MergeRule` to the result and discard mappings that concern its rules from the remaining clone groups, from which we remove all empty and already considered clone groups, in lines 15-18. We repeat this process until no clone groups are left to consider.

```
 1: function ComputeMerge(cl : Cluster[])
 2:     var mergeSpecification = ∅
 3:     for each c ← cl do
 4:         var cg = c.cloneGroups
 5:         while cg ≠ ∅ do      ▷ Create a new sub-cluster
 6:             var top = findTopCloneGroup(cg)
 7:             var mergeRule = createMergeRule(top)
 8:             var considered = {top}
 9:             while hasCompatible(considered, cg) do
10:                 var comp = findTopCompatible(cg)
11:                 var temp = createMergeRule(comp)
12:                 Integrate(mergeRule, temp)
13:                 considered.add(comp)
14:             end while
15:             mergeSpecification.rules.add(mergeRule)
16:             cg.removeMappings(mergeRule.rules)
17:             cg.removeAllEmpty
18:             cg.removeAll(considered)
19:         end while      ▷ Done with current sub-cluster
20:     end for
21:     return mergeSpecification
22: end function
```

Fig. 7. Merge computation.

In the example, considering cluster {*D*, *E*, *F*} containing clone groups CG1, CG2, CG3, and CG5, the largest one CG1 is chosen as top group in line 6. In line 7, a `MergeRule` is created based on CG1, specifying the merge of the involved rules *E* and *F*. One `MergeRuleElement` is created for each pair of clone elements and for each non-clone element, e.g., one for {#15, #20} and one for {#24}. In lines 9-14, CG2 is identified as the next largest compatible clone. Its temporary merge rule, specifying the merge of rules *D*, *E* and *F*, is created. The two merge rules are integrated by establishing that each rule element finally belongs to exactly one `MergeRuleElement`, which involves the deletion of redundant `MergeRuleElement`s. Then, as no compatible clone groups can be found, the `MergeRule` comprising the information of CG1 and CG2 is added to the resulting `MergeSpecification`. In lines 17–18, the mappings of CG3 and CG5 for *D*, *E* and *F* are removed, leaving them empty and leading to their discarding.

Based on a given `MergeSpecification`, the merge refactoring procedure follows Def. 8 (see [19] for a detailed description): non-master-rule elements are moved to the master rule; non-master rules are deleted; a variability model is set for the master rule; and a presence condition is set for each contained element.

6 Evaluation

We focus on two research questions: **RQ1**: *How well does* RuleMerger *achieve its goal of creating high-quality rule sets?* **RQ2**: *What is the impact of design decisions made by* RuleMerger *on the quality of the created rules?*

To answer these questions, we applied our instantiation of *RuleMerger* on rule sets from two real-life model transformation scenarios, called Ocl2Ngc and FmRecog, and one adapted from literature, called Comb. The main quality goal in these scenarios is performance: Ocl2Ngc and Comb were considered as benchmarks in [9] and [23]; FmRecog is an automatically derived rule set used in the context of model differencing [24], a task that necessitates low latency. Thus, we optimized the two input parameters described in Sec. 5 for performance. We describe the rule sets and associated test input models in [19].

We assess the quality of the produced rules with respect to performance and reduction in redundancy. To quantify *performance*, we applied the rule sets on all input models and measured cumulative execution time on all input models. We repeated each experiment ten times to account for variance. To quantify *redundancy reduction*, we measured the relative decrease in the number of rule elements, based on the rationale that we produce semantically equivalent, yet syntactically compacted rules (Theorem 1). As discussed in Sec. 2, reducing redundancy in rules is related to benefits for their maintainability.

6.1 Methods and Set-Up

To address RQ1, we investigated three subquestions: **RQ1.1:** *How do VB rules created by* RuleMerger *compare to the equivalent classical rules?* **RQ1.2:** *How do VB rules created by* RuleMerger *compare to those created manually?* **RQ1.3:** *How do the VB rules created by* RuleMerger *scale to large input models?* For RQ1.1, we considered all three rule sets. For RQ1.2, we considered the scenario

where a manually created rule set was available: Ocl2Ngc [9]. For RQ1.3, we considered the Comb scenario, as it features a procedure to increase the input model automatically (increasing the size of the input grid [23]); we measured the impact of model size on execution time until we ran out of memory.

To address RQ2, we investigated two questions: **RQ2.1** *What is the impact of clone detection?* **RQ2.2** *What is the impact of clustering?* For RQ2.1, we randomly discarded 25%–100% of the reported clone groups. For RQ2.2, we replaced the default clustering strategy by one that assigns rules to clusters randomly. We measured the execution time of the rules created using the modified input.

As clone detection techniques, we applied ConQat [22] on Ocl2Ngc and FmRecog, as it was the only tool scaling to these scenarios. We applied gSpan [20] on the Comb rule set as it allowed us to consider all clones instead of an approximation. The input parameters were optimized independently for each scenario by applying the technique repeatedly until the execution time was minimized. Moreover, the Henshin transformation engine features an optimization concerning the order of nodes considered during matching. To avoid biasing the performance of the FmRecog rule set by that optimization, we deactivated it. We ran all experiments on a Windows 7 workstation (3.40 GHz processor; 8 GB of RAM).

6.2 Results and Discussion

Table 2 shows the size and performance characteristics for all involved rule sets. Execution time is provided in terms of the total and median amount of time required to apply the whole rule set on each test model, each of them paired with the standard deviation *(SD)*. The number of elements refers to edges and nodes, including both left-hand and right-hand side of the involved rules.

RQ1.1 The execution time observed for Ocl2Ngc after the *RuleMerger* treatment showed a decrease by the factor of 158. This substantial speed-up can be partly explained by the merging component of *RuleMerger* that eliminates the anti-pattern *Left-hand side not connected (LhsNC)* [18]: In the automatically constructed VB rules, connected rules are used as base rules, while in the classic rules, we found multiple instances of *LhsNC*. In the FmRecog and Comb rule sets, the speed-up was less drastic, amounting to the factors of 4.5 and 5.8, respectively. When applying the Comb rule set on the SeveralMatches sce-

		Size		*Execution time (sec.)*			
Scenario	*Rule Set*	#Rules	#Elements	Total	Sd	Median	Sd
Ocl2Ngc	Classic	36	3045	916.6	96.3	46.0	7.1
	Manual Merge	10	1018	181.8	27.1	10.8	2.4
	RuleMerger	12	2147	5.8	0.4	0.4	0.1
FmRecog	Classic	53	4626	799.9	41.4	63.2	3.5
	RuleMerger	12	2790	211.4	46.0	15.9	0.3
Comb	Classic	6	252	1.39	0.09	0.12	0.01
NoMatch	*RuleMerger*	1	62	0.24	0.09	0.02	0.01
Comb	Classic	6	252	10.4	0.18	0.83	0.02
SeveralMatches	*RuleMerger*	1	62	14.2	0.26	1.07	0.05

Table 2. Results for RQ1.1 and RQ1.2: Quality characteristics of the rule sets.

nario, which involves an artificial input model with many possible matches [23], execution time increased by the factor 1.36, showing a limitation of VB rules: If the number of base matches is very high, the initialization overhead for extending the base matches outweighs the initial savings. This overhead may be reduced by extending the transformation engine implementation. The amount of redundancy was reduced by 29% in Ocl2Ngc, 40% in FmRecog, and 75% in Comb.

RQ1.2 In Ocl2Ngc, we found a speed-up by the factor of 36. To study this observation further, we inspected the manually created rules, again finding several instances of the *LhsNC* antipattern. This observation gives rise to an interesting interpretation of the manual merging process: While the designer's *explicit* goal was to optimize the rule set for performance, they implicitly performed the more intuitive task of optimizing for compactness. Indeed, the amount of reduced redundancy in the manually created rules (67%) was significantly greater than in those created by *RuleMerger* (29%), highlighting an inherent trade-off between performance- and compactness-oriented merging: Not including overlap elements into the base rule leads to duplications in the variable portions.

RQ1.3 As shown in Fig. 8, the last supported input model was a 480x480 grid for both rule sets. We observed that the ratio between the execution time of applying the classic (dark-gray bars) and the VB rules (light-gray bars) stayed the same in each iteration, independent of the size of the input grid: The VB rules were always faster by the factor of 6. In terms of the total execution time, the speed-up provided by the VB rules became more important as the size of input models increased.

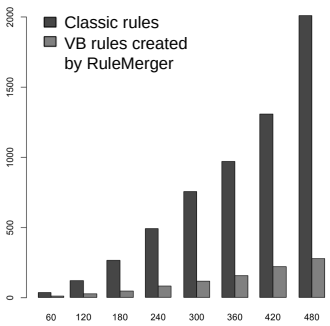

Fig. 8. Results for RQ1.3: Execution time in sec. (y) related to length of grid (x).

RQ2.1 As presented in Table 3, the execution time for the FmRecog rule set increased monotonically when we increased the amount of discarded overlap, denoted as d. Ocl2Ngc behaved almost monotonically as well. The slightly decreased execution time reported for d=0.25 can be explained by the heuristic merge construction strategy. While the merge of rules based on their largest clones might be adequate in general, in some cases it may be preferable to discard a large clone in favor of a more homogeneous distribution of rules. The reported execution time for d=0.75 was higher than that for the set of classic rules. In this particular case, small clones were used during merging, leading to small base rules, which resulted in many detectable matches and thus in a high initialization overhead for extending these matches. To mitigate this issue, one could define a lower threshold for clone size.

RQ2.2 As indicated in Table 4, the employed clustering strategy had a significant impact on performance, amounting to factors of 13.7 for the Ocl2Ngc and 3.7 for the FmRegoc rule set. Interestingly, in Ocl2Ngc, random clustering still yielded better execution times than manual clustering did (see Table 2) – this is related to the fact that *RuleMerger* removed the *LhsNC* antipattern. In FmRecog, randomly clustered rules were comparable to the classic ones.

	d: Discarded portion				
Scenario	0.0	0.25	0.5	0.75	1.0
Ocl2Ngc	5.8	5.6	251	981	917
FmRecog	211	252	604	690	800

Table 3. Results for RQ2.1: Impact of considered overlap on execution time (sec.).

	Clustering strategy	
Scenario	AvLinkage	Random
Ocl2Ngc	5.8	80
FmRecog	211	788

Table 4. Results for RQ2.2: Impact of clustering strategy on execution time (sec.).

6.3 Threats to validity and limitations

Factors affecting *external validity* include our choice of rule sets, test models and matching strategy, and the capability to optimize the two input parameters. While the considered rule sets represent three heterogeneous use cases, examples to show that our approach scales to more diverse and larger scenarios are required. To ensure that our test models were realistic, we employed the original test or benchmark models. The performance of rule application depends on the chosen matching strategy, in our case, mapping this task to a constraint satisfaction problem [25]. We aim to consider the effect of other strategies in the future. Parameter tuning requires the existence of realistic test models. If a rule set is designed for productive use, it is reasonable to assume such models to exist.

With regard to *construct validity*, we focus on one aspect of maintainability, the amount of redundancy. Giving a definitive answer on how to unify rules for their optimal maintainability is outside the scope of this work. Specifically, several unrelated rules may be unified, impairing understandability. To mitigate this issue, we recommend to inspect the clustering result before merging. Furthermore, our approach increases the size of individual rules, a potential impediment to readability [26]. We believe that this limitation can be mitigated by tool support. Inspired by related approaches to address the readability issues associated with *#ifdef* directives [27,28], we aim to provide *editable views*, representing portions of a VB rule that correspond to user-selected configurations.

7 Related Work

Our work is related to a number of approaches that create feature-annotated representations of products lines. In [29], an approach to merge statecharts based on structural and behavioral commonalities is applied to models of telecommunication features. In [16], an approach for merging and identifying variability in Matlab product variants is proposed. In [14,30], a formal merge framework is defined and instantiated for class models and state machines. It is studied how a number of desired qualities of the resulting model can be obtained. In [31,32], a technique for the reverse engineering of variability from block diagrams based on their data-flow structures is introduced. In [15], a language-independent approach for the reverse-engineering of product lines is proposed. These approaches operate on the basis of an element-wise comparison using names and as well as structural and behavioral similarities. In model transformation rules, the essential information lies in isomorphic structural patterns. To our knowledge, our approach is the first that utilizes clone detection to identify such patterns.

Our work can be considered a performance optimization for the NP-complete problem of transformation rule matching [33]. Earlier approaches in this area are mostly complementary to ours as they focus on the matching of single rules [34,35,36,37]. Mészáros et al. [38] first explored the idea of considering overlapping portions in multiple rules. Their custom technique for detecting these portions, however, did not scale up to complete rule sets. Instead, it considers just two rules at once, enabling a moderate performance improvement of 11%. In our approach, applying clone detection and clustering techniques gives rise to an increased speed-up. In [39], shared sub-patterns are considered dynamically during incremental pattern matching to mitigate the memory issue of Rete networks. Yet, the authors report on deteriorated execution times: The index tables mapping sub-patterns to partial matches grow so large that performance is impaired. Multi-query optimization has also been investigated in relational databases [40]. In graph databases, only single-query optimization has been considered [41].

The maintainability effects of cloning have been studied intensively [42,14]. In an empirical study, Kim et al. [43] identified three types of clones: *short-lived clones* vanishing over the course of few revisions, *"unfactorable" clones* related to language limitations, and *repeatedly changing clones* where a refactoring is recommended. We second the idea that an aggressive refactoring style directed at short-lived clones should be avoided. Instead, targeting clones of the two latter categories, we propose to apply our approach to stable revisions of the rule set. Specifically, clones that were previously "unfactorable" due to the lack of suitable reuse concepts may benefit from the introduction of VB rules. An approach complementary to clone refactoring is *clone management*, based on a tool that detects and updates clones automatically [44]. This approach has a low initial cost, but requires constant monitoring. Further works propose the refactoring of transformation rules towards pre-defined patterns [45], modular interfaces [46], and abstract metamodels [47]. None of these considers clones.

8 Conclusion and Future Outlook

In this work, we introduced an approach for constructing variability-based (VB) model transformation rules automatically. Our experiments showed that the approach is effective: The created rules always had preferable quality characteristics when compared to classical rules, unless the number of expected matches was very high. It is apparent that using the approach, the performance of model transformation systems as well as redundancy-related maintainability concerns can be considerably improved, making the benefits of VB rules available while imposing little manual effort.

In the future, we aim to provide tool support to address the readability issue brought by the increased amount of information in each rule. Moreover, we plan to increase the expressiveness of VB rules. Covering all important transformation features such as application conditions and amalgamation will make VB rules applicable to the existing variety of model transformation languages [48,49,50].

Acknowledgements: We thank Felix Rieger and the anonymous reviewers for their valuable comments on the present and earlier drafts of this manuscript.

References

1. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software **20**(5) (2003) 42–45
2. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal **45**(3) (2006) 621–646
3. Hermann, F., Gottmann, S., Nachtigall, N., Braatz, B., Morelli, G., Pierre, A., Engel, T.: On an Automated Translation of Satellite Procedures using Triple Graph Grammars. In: ICMT'15. Springer (2013) 50–51
4. Mann, M., Ekker, H., Flamm, C.: The graph grammar library-a generic framework for chemical graph rewrite systems. In: ICMT'13. Springer (2013) 52–53
5. Famelis, M., Lucio, L., Selim, G., Salay, R., Chechik, M., Cordy, J.R., Dingel, J., Vangheluwe, H., S., R.: Migrating Automotive Product Lines: A Case Study. In: ICMT'15. Springer (2015)
6. Richa, E., Borde, E., Pautet, L.: Translating ATL Model Transformations to Algebraic Graph Transformations. In: ICMT'15. Springer (2015) 183–198
7. Kusel, A., Schonbock, J., an G. Kappel, M.W., Retschitzegger, W., Schwinger, W.: Reuse in Model-To-Model Transformation Languages: Are We There Yet? Software & Systems Modeling **14** (2013) 537–572
8. Blouin, D., Plantec, A., Dissaux, P., Singhoff, F., Dignet, J.P.: Synchronization of Models of Rich Languages with Triple Graph Grammars: an Experience Report. In: ICMT'14. (2014)
9. Strüber, D., Rubin, J., Chechik, M., Taentzer, G.: A Variability-Based Approach to Reusable and Efficient Model Transformations. In: FASE'15, Springer (2015) 283–298
10. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Ser. in SE. Addison-Wesley (2001)
11. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05, Springer (2005) 422–437
12. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Science of Computer Programming **74**(7) (2009) 470–495
13. Xu, R., Wunsch, D., et al.: Survey of Clustering Algorithms. IEEE Trans. on Neural Networks **16**(3) (2005) 645–678
14. Rubin, J., Chechik, M.: Combining Related Products into Product Lines. In: FASE'12, Springer (2012) 285–300
15. Ziadi, T., Henard, C., Papadakis, M., Ziane, M., Le Traon, Y.: Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines. In: SAC'14, ACM (2014) 1064–1071
16. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Automatic Variation-Point Identification in Function-Block-Based Models. In: GPCE'10, ACM (2010) 23–32
17. Fowler, M.: Refactoring: improving the design of existing code. Pearson Education India (2002)
18. Tichy, M., Krause, C., Liebel, G.: Detecting Performance Bad Smells for Henshin Model Transformations. AMT'13 **1077** (2013)
19. Strüber, D.: Model-Driven Engineering in the Large: Refactoring Techniques for Models and Model Transformation Systems. PhD thesis, Philipps University Marburg (2016) pending publication.
20. Yan, X., Han, J.: gspan: Graph-Based Substructure Pattern Mining. In: ICDM'03, IEEE (2002) 721–724
21. Pham, N.H., Nguyen, H.A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N.: Complete and Accurate Clone Detection in Graph-Based Models. In: ICSE'09, IEEE (2009) 276–286
22. Deissenboeck, F., Hummel, B., Juergens, E., Pfaehler, M., Schaetz, B.: Model Clone Detection in Practice. In: the 4th International Workshop on Software Clones, ACM (2010) 57–64
23. Varró, G., Schürr, A., Varró, D.: Benchmarking for Graph Transformation. In: ISVLHCC'05, IEEE (2005) 79–88

24. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about Product-Line Evolution using Complex Differences on Feature Models. Journal of ASE (2015)
25. Rudolf, M.: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: Wksp. on Theory and Application of Graph Transformations, Springer Science & Business Media (1998) 238
26. Störrle, H.: On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters. In: MoDELS'14. Springer (2014) 518–534
27. Kästner, C.: Virtual separation of concerns. PhD thesis, University of Magdeburg (2010)
28. Walkingshaw, E., Ostermann, K.: Projectional editing of variational software. In: GPCE'14, ACM (2014) 29–38
29. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Variant Feature Specifications. IEEE TSE **38**(6) (2012) 1355–1375
30. Rubin, J., Chechik, M.: Quality of Merge-Refactorings for Product Lines. In: FASE'13, Springer (2013) 83–98
31. Wille, D.: Managing lots of models: the famine approach. In: FSE'14, ACM (2014) 817–819
32. Holthusen, S., Wille, D., Legat, C., Beddig, S., Schaefer, I., Vogel-Heuser, B.: Family model mining for function block diagrams in automation software. In: SPLC'14: Workshops, Demonstrations and Tools Companion, ACM (2014) 36–43
33. Atallah, M.: Algorithms and Theory of Computation Handbook. CRC (2002)
34. Varró, G., Friedl, K., Varró, D.: Adaptive Graph Pattern Matching for Model Transformations using Model-Sensitive Search Plans. ENTCS **152** (2006) 191–205
35. Horváth, Á., Varró, G., Varró, D.: Generic Search Plans for Matching Advanced Graph Patterns. Elec. Comm. of the EASST **6** (2007)
36. Krause, C., Tichy, M., Giese, H.: Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In: FASE'14, Springer (2014) 325–339
37. Acretoaie, V., Störrle, H.: Efficient Model Querying with VMQL. In: CMSEBA'14, CEUR-WS.org (2015) 7–16
38. Mészáros, T., Mezei, G., Levendovszky, T., Asztalos, M.: Manual and automated performance optimization of model transformation systems. International Journal on Software Tools for Technology Transfer **12**(3-4) (2010) 231–243
39. Varró, G., Deckwerth, F.: A Rete Network Construction Algorithm for Incremental Pattern Matching. In: ICMT'13. (2013) 125–140
40. Sellis, T.K.: Multiple-query optimization. ACM Transactions on Database Systems (TODS) **13**(1) (1988) 23–52
41. Zhao, P., Han, J.: On graph query optimization in large networks. VLDB Endowment **3**(1-2) (2010) 340–351
42. Kapser, C., Godfrey, M.W.: "cloning considered harmful" considered harmful. In: Working Conference on Reverse Engineering, IEEE (2006) 19–28
43. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: ACM SIGSOFT Software Engineering Notes. Volume 30., ACM (2005) 187–196
44. Nguyen, H.A., Nguyen, T.T., Pham, N.H., Al-Kofahi, J., Nguyen, T.N.: Clone management for evolving software. Software Engineering, IEEE Transactions on **38**(5) (2012) 1008–1026
45. Syriani, E., Gray, J.: Challenges for Addressing Quality Factors in Model Transformation. In: ICST'12, IEEE (2012) 929–937
46. Rentschler, A.: Model Transformation Languages with Modular Information Hiding. PhD thesis, Karlsruher Institut für Technologie (2015)
47. Cuadrado, J.S., Guerra, E., de Lara, J.: Reverse Engineering of Model Transformations for Reusability. In: ICMT'14, Springer (2014) 186–201
48. Balasubramanian, D., Narayanan, A., van Buskirk, C., Karsai, G.: The Graph Rewriting and Transformation Language: GReAT. ECEASST **1** (2007)
49. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: ICGT'06, Springer (2006) 383–397
50. Acretoaie, V., Störrle, H., Strüber, D.: Transparent Model Transformation: Turning Your Favourite Model Editor into a Transformation Tool. In: ICMT'15, Springer (2015)