Under consideration for publication in Formal Aspects of Computing

# Variability-based model transformation: formal foundation and application

D. Strüber[1,2], J. Rubin[3,4], T. Arendt[2,5], M. Chechik[6], G. Taentzer[2], J. Plöger[2]

[1] University of Koblenz and Landau, Koblenz, Germany,
[2] University of Marburg, Marburg, Germany,
[3] Massachusetts Institute of Technology, Cambridge, USA,
[4] University of British Columbia, Vancouver, Canada,
[5] GFFT Innovationsförderung GmbH, Bad Vilbel, Germany,
[6] University of Toronto, Toronto, Canada

**Abstract.** Model transformation systems often contain transformation rules that are substantially similar to each other, causing maintenance issues and performance bottlenecks. To address these issues, we introduce *variability-based model transformation*. The key idea is to encode a set of similar rules into a compact representation, called *variability-based rule*. We provide an algorithm for applying such rules in an efficient manner. In addition, we introduce rule merging, a three-component mechanism for enabling the automatic creation of variability-based rules. Our rule application and merging mechanisms are supported by a novel formal framework, using category theory to provide precise definitions and to prove correctness. In two realistic application scenarios, the created variability-based rules enabled considerable speedups, while also allowing the overall specifications to become more compact.

## 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 [57]. Algebraic graph transformation (AGT) is one of the main paradigms in model transformation. It allows the specification of rules in a high-level, declarative manner [17]. Recently, AGT has been used to describe and implement complex transformations in various domains, such as aerospace engineering, [28], chemistry [39], and automotive software product lines [23]. AGT is gaining further importance due to its use as an analysis back-end for imperative model transformation languages [50].

Transformation systems often contain rules that are substantially similar to each other. Yet, various model transformation languages lack constructs suited to capture these similar *rule variants* in a compact manner [37, 61]. The most frequently applied mechanism for creating variants remains cloning: developers produce 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: rule sets are often applied in *batch mode* – all rules are considered as long as one of them is applicable. In this scenario, 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]. Later in this paper, we consider selected rule sets from real-life scenarios in the domains of model versioning and constraint translation, in which these maintainability and performance drawbacks manifested themselves.

*Correspondence and offprint requests to*: Daniel Strüber, University of Koblenz-Landau, Institute for Computer Science, Universitätsstr. 1, 56070 Koblenz, Germany. e-mail: strueber@uni-koblenz.de

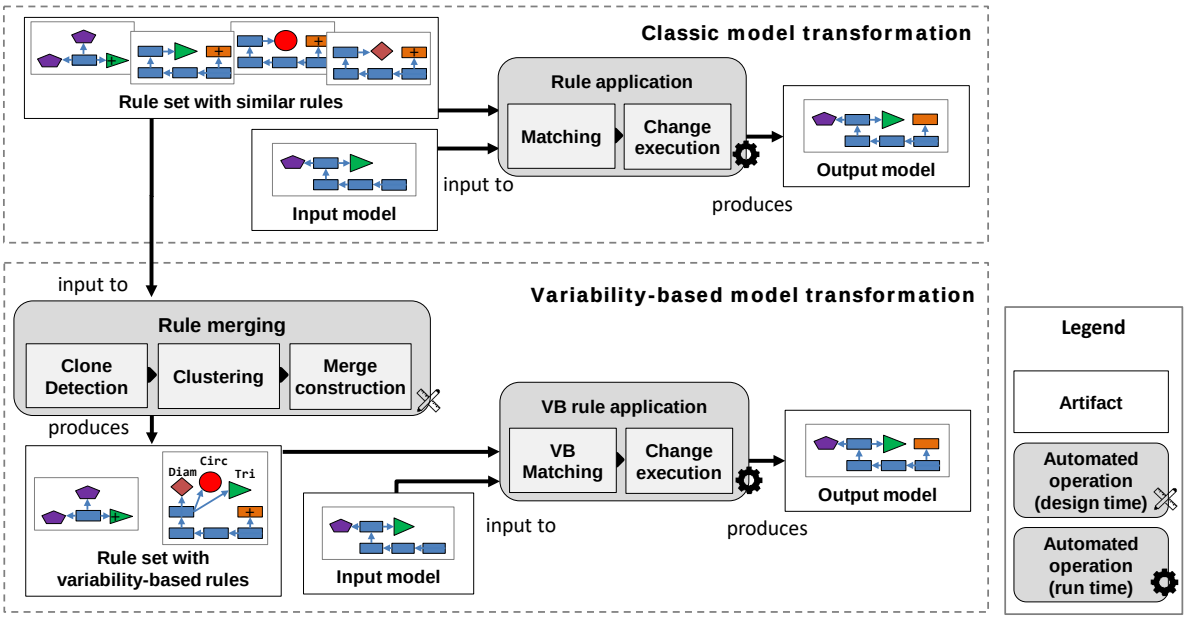


**Fig. 1.** Overview.

In this paper, we present *variability-based model transformation*, an approach inspired by product line engineering (PLE) principles [12, 16]. As illustrated in Fig. 1, we introduce the concept of variability-based (VB) rules for encoding a set of similar rules into a single-copy representation, explicating their common and variable portions. By introducing VB rules into a rule set, the maintainability and performance drawbacks of cloning are addressed as follows:

- Representing a set of mutually similar rules in a single-copy representation can improve their *maintainability*: it is much easier to maintain consistency between rules if the developer is not required to perform the same change in each rule individually. This approach is also less prone to subtle errors introduced during rule creation by cloning, where errors are copied and must be fixed in all subsequently created rules. Finally, the effort to maintain the overall rule set can be smaller when the overall number of rules is also smaller, in particular, if the differences between the input rules are subtle, so that the common representation does not grow too large.
- To improve the *performance* in scenarios where all represented rules are to be applied, we replace the classic rule application mechanism with a variability-based one. *VB rule application* features a novel algorithm for *matching*, i.e., finding application sites in the input model. Our key idea is to find matches for common parts of the represented rules first and then to use them as starting points to match the variable parts. Afterwards, changes specified by the rules, such as the creation of new elements, are executed in the classic way. With VB matching, we address the computational bottleneck of rule application: matching entails the NP-hard *sub-graph isomorphism* problem [13].

To make these benefits available to existing rule sets, rules need to be unified to become VB rules. When performed manually, this is a tedious and error-prone task relying on the precise identification of (i) sets of rules that should be unified into a single VB rule; (ii) rule portions that should be merged versus portions that should remain separate. We introduce *rule merging* as an approach to automate this task. Rule merging comprises three components: It applies *clone detection* to identify overlapping fragments, *clustering* to group mutually similar rules, and *merge construction* to create VB rules using this information. Rule merging can be performed at design time, before the rules are applied.

**Contributions.** This paper is an extended version of our earlier work [63, 64], improving it in two main respects. First, it replaces the earlier operational description of our approach with a precise formal foundation in category theory. The new foundation enables greater flexibility w.r.t. the supported transformation languages, since it allows arbitrary graph categories to be used as base graphs; for instance, meta-model conformance and attributes can be represented using typed attributed graphs [27]. In addition, it serves as a prerequisite for our second contribution: we extend our approach to negative application conditions (NACs), a main feature of graph-based model transformations [26]. A NAC prevents its host rule from being applied whenever a specified graph structure is present in the input model. NACs have not been addressed in our earlier work since they are particularly challenging to handle during rule application and merging: a

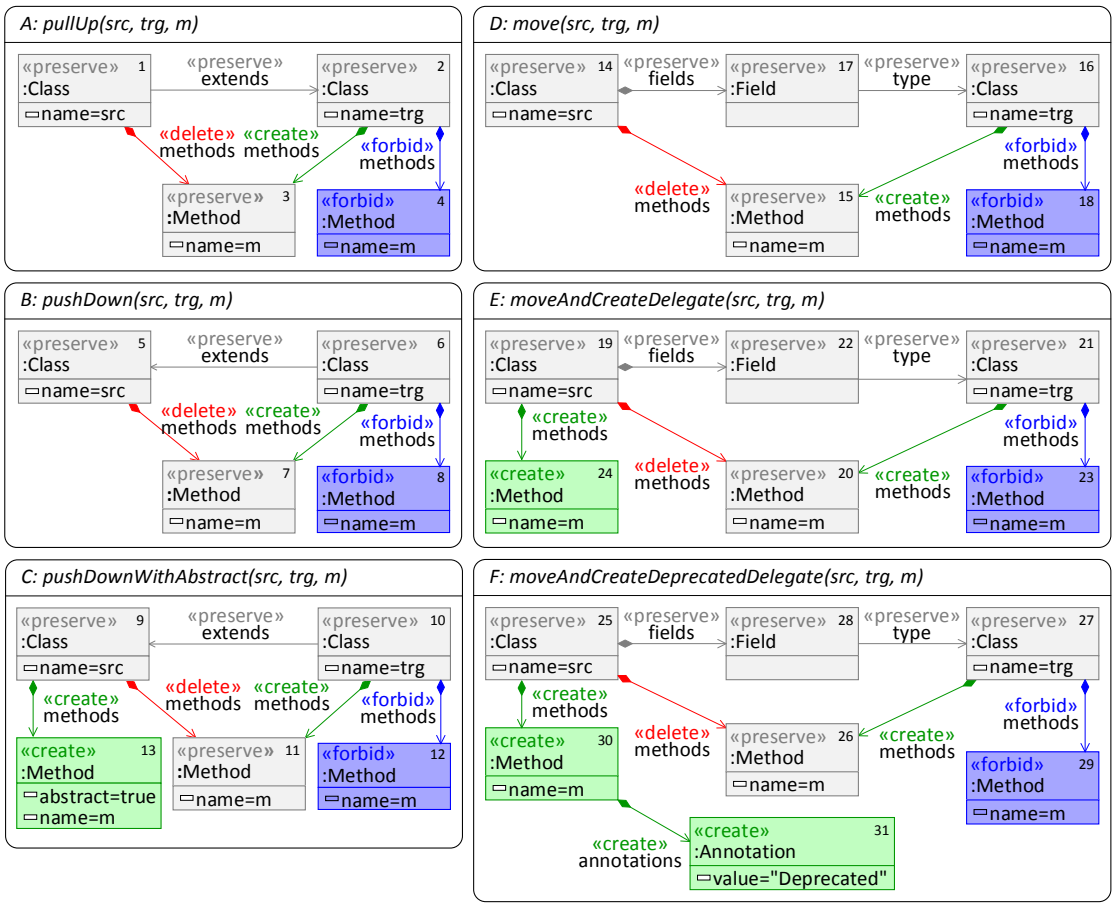


**Fig. 2.** Refactoring rules for class models.

particular NAC may be satisfied when considering it in the full context of its host rule, while *not* being satisfied when considering only parts of the NAC or its host rule.

Specifically, we make the following contributions:

- We provide a formalization of variability-based rules, expressing their syntax and application semantics on the basis of graph transformation. We prove equivalence to the application of the corresponding classic rules.
- We present a formal framework for the merging of rule sets with similar rules, comprising clone detection, rule clustering, and merge construction steps. We formally show the equivalence of the produced VB rules to their classical counterparts.
- We present algorithms for the automated computation of merging and rule application results. In combination, both algorithms aim to achieve a performance gain compared to matching and applying the classical input rules individually.
- We empirically show that the produced VB rules are superior to their classical counterparts in terms of several performance- and maintainability-related characteristics.

The remainder of this paper is structured as follows: In Sec. 2, we demonstrate our approach in two motivating scenarios. In Sec. 3, we recall the necessary background required by the approach. In Sec. 4, we define the foundations of variability-based model transformation. We provide results to facilitate correctness of our rule merging and application mechanisms. Sec. 5 reports on the implementation of our approach for Henshin, a graph-based model transformation language [4]. Sec. 6 presents our evaluation. In Sec. 7 and 8, we discuss related work and conclude, respectively.

## 2. Motivating examples

### 2.1. Variability-intensive refactoring rules

Consider a set of model transformation rules aiming to improve the structure of an existing code base using *class model refactorings*. Fig. 2 shows six refactoring rules expressed in an abstract syntax notation [17]. The rules describe several ways of relocating a method between two classes. Each rule comprises three parts called left-hand side (LHS), negative application conditions (NACs), and right-hand side (RHS): if the LHS matches a place in the input model and all NACs are fulfilled, i.e., cannot be matched, then the RHS is applied, thereby changing the model. We present the rules in an integrated form, with the LHS, NACs, and RHS of a rule being represented in one graph. The LHS comprises all *delete* and *preserve* objects. NACs correspond to *forbid* objects; if a rule has multiple NACs, they are distinguished by index numbers. The RHS 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. This is possible only if the super-class does not already contain a method with the specified name. 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 method as deprecated. Each rule has a NAC preventing the method from being relocated if the target class already has a method of the same name. In absence of a suitable concept to capture similar rule variants, there is no choice but to create such rule sets by cloning, that is, copying a 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 input 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 mutual exclusion: *Rule 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 #40 and its incoming edge are removed in the configuration {*C=false*, *B=true*}. Elements without a visible presence condition, such as nodes #32-#34, are actually annotated with *true*; we omit this presence condition from the figures for simplicity. The NACs of *Rules A–F* are merged to just one per VB rule.

A variability-based rule is equivalent to a set of rules for all its valid configurations. Yet this example demonstrates several benefits of VB rules related to maintainability: The amount of redundancy is reduced, ensuring consistency between variants during changes; subtle errors produced during rule creation are fixed in one place. The total number of rules is smaller, potentially allowing to navigate the transformation system with less effort and decreased computational cost for the rule editor. The latter may increase responsiveness of the editor and thereby help the developer focus on the maintenance task at hand.

In this example, the VB rules are configured individually, either manually by the user or automatically by client code, e.g., a model editor. The result of the configuration is a "flat" rule again – a process similar to that in product-line engineering approaches [16]. Alternatively, *all* rules of a rule set may be executed in *batch mode*, i.e., considered simultaneously, as we demonstrate below.

### 2.2. Standardization enforcement in state machines

This example focuses on standardization enforcement in state machines. The overall purpose of this transformation is to ensure conformance with the UML standard for state machines [43]. In UML, states can have entry and exit actions that are executed whenever the state becomes active or inactive, respectively. We assume that the input state machine comes from a different state machine notation where transitions between states can have actions as well. In the example state machine of Fig. 4, this is the case for the transitions labelled with the actions `wash.Start()` and `QuickCool()`. Therefore, the task is to *"fold"* these actions, so that common actions of incoming or outgoing transitions of the same state are moved to the state. This transformation has been introduced as an example in [11].

Fig. 5 shows two model transformation rules for establishing this goal. Again, we present the rules in an integrated form. Via its left-hand side, i.e. nodes and edges with *preserve* and *delete* annotations, *Rule G* searches for a pattern

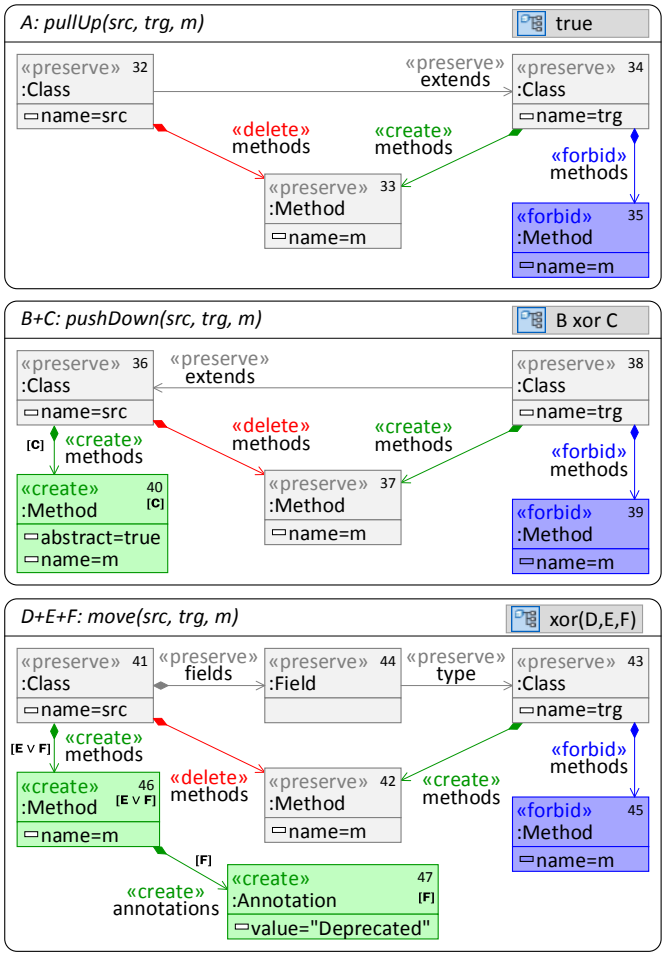


**Fig. 3.** Variability-based class model refactoring rules.

where a state has two incoming transitions being labelled with the same action. If such a pattern is present, the action is moved from the transitions to the target state to become its `entryAction`; this moving is expressed in terms of a parallel deletion and creation of edges between the action and its host elements. As indicated by the annotations *forbid#1* and *forbid#2*, the rule comprises two NACs. These NACs prevent the rule from being applied when the target state already has an entry action (forbid#1) or when it has more than two incoming transitions (forbid#2); both cases are conflict situations that have to be resolved with developer input. *Rule G* is the dual counterpart of *Rule H*; its purpose is to move a common action of two outgoing transitions of the same state to this state, thus making it the `exitAction` of this state. Please note that the general case allows moving the action if it is present in *all* incoming and outgoing transitions; we only consider two transitions for simplicity. To support the general case, *Rules G* and *H* can be generalized to $n$ incoming and outgoing transitions using *multi-amalgamation* [25], an advanced concept of algebraic graph transformations.

Unlike the example transformation in Sect. 2.1, this transformation is executed in batch mode, i.e., by applying both rules to the state machine as long as one of them is applicable. When this rule set is applied to larger state machines, an unnecessary performance overhead can arise: even though both rules share a considerable overlap part, including all *preserve* and *delete* nodes, the classic transformation approach handles them separately. As a prerequisite for enabling a faster execution, one can explicate this overlap part, resulting in the VB rule shown in Fig. 6. Similar to the previous example, to fully represent the original rules, the individual parts of both rules are captured using *variation points*, corresponding to the rule names. Rule elements are *annotated* with *presence conditions* over these variation points. Again, for the simplicity of presentation, we hide the presence condition *true*, e.g., for nodes #65-#68. The overall VB rule has four NACs, representing the two NACs of *Rule G* and the two NACs of *Rule H*.

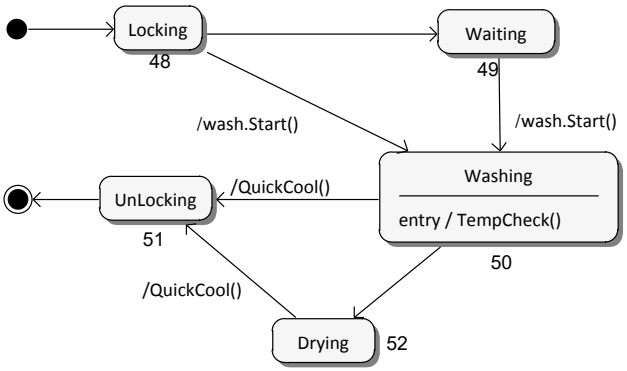


**Fig. 4.** Example state machine.

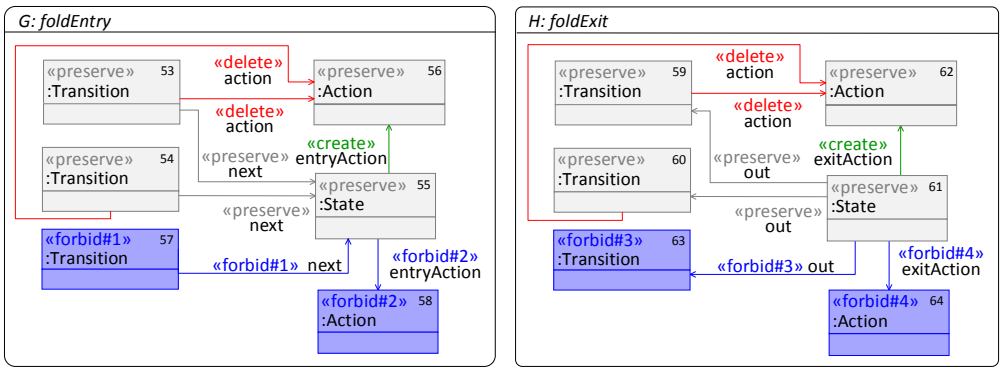


**Fig. 5.** Standardization enforcement rules for state machines.

In such batch transformation scenarios, the match-finding algorithm in VB rule application performs matching of all valid configurations at once, aiming to positively affect the performance of the transformation system. To detect configurations leading to applicable rules, a two-step process is applied. First, the *base rule* is matched – the portion of the rule representing common parts of all individual rules, i.e., those elements annotated with *true*. For the state machine example, the base rule comprises elements #65–68 and the *delete* edges. This matching process produces a number of *pre-matches*, i.e., mappings between the rule and the input model. In our example, one of these pre-matches, called $p_{base}$, assigns the base rule to element #51 with its incoming transitions and their common action. NACs are not checked during this first step. Second, if the set of pre-matches is non-empty, we enumerate the valid configurations and search matches for the resulting rules by extending the pre-matches. In the course of this matching step, we also consider the NACs of the resulting rules. In the example, this yields exactly one match: $m_{\mathrm{G}}$, an extension of $p_{base}$ that incorporates the *preserve* edges specified by *Rule G*. Another potential match, mapping the base rule to #50 and its incoming transitions, is ruled out because it does not satisfy NAC 2: the `Washing` state already has an entry action. The matching result is $m_{\mathrm{G}}$ paired with the configuration {*G=true*; *H=false*} allowing $m_G$ to exist.

## 3. Preliminaries

As preliminaries for our approach, we revisit the use of algebraic graph transformation as a formal foundation for model transformations as presented in e.g. [20]. Crucial for us is the notion of *graph*, comprising a set of nodes and a set of directed edges connecting these nodes. Structure-compatible mappings between graphs can be expressed in terms of *graph morphisms* which are compatible with the source and target functions for the edges. Graphs and graph morphisms can be used to represent the underlying structure of visual models: nodes, edges, and graph morphisms represent model elements, references between model elements, and relationships between models, respectively.

In category theory, graphs together with graph morphisms form a *category*, since graph morphisms can be composed associatively and for each graph an identity graph morphism exists. Graphs with additional “features”, such

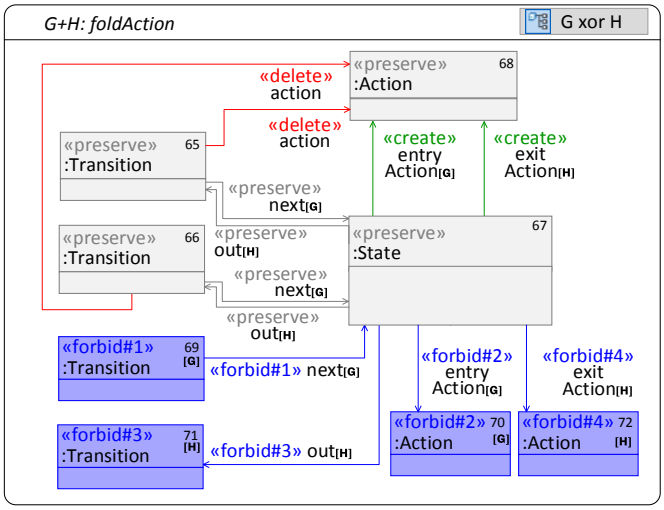


**Fig. 6.** Variability-based *fold* rule.

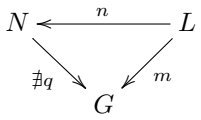


**Fig. 7.** Negative application condition.

as typing and attributes, in many cases form categories as well as shown in e.g. [27, 20]. Our formal foundation for variability-based transformation is flexible in the sense that it leaves the chosen category of graphs open, allowing a suitable one to be inserted. For instance, the notion of meta-model conformance can be represented in terms of a distinguished graph, called *type graph*, and suitable typing morphisms between graphs and the type graph. In our implementation and evaluation, we use the full power of typed attributed graphs with inheritance [27], since these graph features are orthogonal to variability.

Graphs and graph morphisms are the building blocks of algebraic graph transformations. Specifically, we consider the rule-based transformation approach called *double-pushout approach* (DPO). In this approach, graph elements occurring in the left and right-hand sides of a rule, i.e., in an *interface graph*, are used to glue new elements to existing ones. Rules may have *negative application conditions* [26], making the rule inapplicable in case a certain graph structure is present.

**Definition 1 (Rule with negative application conditions).** A rule $r = (L \xleftarrow{le} I \xrightarrow{ri} R, NAC)$ consists of graphs $L$, $I$ and $R$, called *left-hand side*, *interface graph* and *right-hand side*, respectively, two injective graph morphisms $le$ and $ri$, and *NAC*, a set of negative application conditions on $L$.
A *negative application condition* $n \in NAC$ on a graph $L$ is a graph morphism $n : L \to N$. A graph morphism $m : L \to G$ *satisfies* $n$, written $m \models n$, if $\nexists q : N \to G$ such that $q$ is a graph morphism and $m = q \circ n$ (see Fig. 7).
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 $G$.

In the refactoring example, the rules in Figs. 2 and 5 follow this definition. Elements of $I$ are annotated with *preserve*, elements of $L \setminus le(I)$ with *delete*, and elements of $R \setminus ri(I)$ with *create*. Each rule in Fig. 2 has one NAC, with the graph $N$ being made up by all elements marked *forbid*, *preserve*, or *delete*. The two rules in Fig. 5 have two NACs each: the first one of *Rule G* comprises the *forbid#1*, *preserve*, and *delete* elements; the other one incorporates the *forbid#2* elements instead of the *forbid#1* ones. In both figures, all rules are connected.

To apply a rule to a given graph $G$, its left-hand side $L$ needs to be matched to $G$. Formally, this matching can be expressed via a graph morphism $m$ called *match*, which maps each element in $L$ to a counterpart in $G$. The application of a rule consists of two steps. First, all graph elements in $m(L \setminus le(I))$ are deleted. Nodes to be deleted may have adjacent edges which have not been matched, so the rule application may produce dangling edges. Therefore, all matches $m$ have to satisfy the *gluing condition*: if a node $n \in m(L)$ is to be deleted by the rule application, it has to delete all adjacent edges as well. Afterwards, unique copies of $R \setminus ri(I)$ are added. The DPO approach is based on the

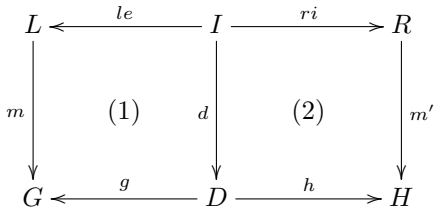


**Fig. 8.** Rule application by a double pushout (DPO).

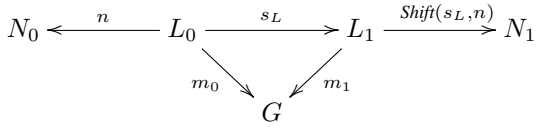


**Fig. 9.** Shifting negative application condition $n$ over graph morphism $s_L$.

observation that this behavior can be characterized by a double-pushout [19]. Given a rule and a match, the resulting rule application is unique [19].

**Definition 2 (Rule application).** Let a graph $G$, a rule $r = (L \xleftarrow{le} I \xrightarrow{ri} R, \mathit{NAC})$, and a total graph morphism $m : L \to G$ be given. A *rule application* from $G$ to a graph $H$, written $G \Rightarrow_{r,m} H$, is given by the diagram in Fig. 8, where (1) and (2) are pushouts, and $m$ satisfies each NAC in *NAC*, written $m \models \mathit{NAC}$. We refer to $G$, $m$ and $H$ as a *start graph*, a *match*, and a *result graph*, respectively.

In the state machine example, *Rule G* can be matched to the input state machine via match $m_G$, comprising mappings of elements #65–68 to element #51, its incoming transitions, and their common action. Considering element #50 instead would not yield a match, since *NAC2* is not fulfilled: state *Washing* already has an entry action. By the application of *Rule G*, the action is removed from the incoming transitions by deleting their connecting edges. As no dangling edges are left behind, the gluing condition is satisfied. An edge between the #51 and the action is created, yielding a state machine where *UnLocking* has the entry action `QuickCool()`.

Next we consider an operation that allows shifting NACs between rules over a morphism [21, 22]. Per [21], this construction exists for all NACs.

**Definition 3 (Shift).** Given a NAC $n$ over $L_0$ and a morphism $s_L : L_0 \to L_1$, *Shift* is an operation that transforms $n$ via $s_L$ into an application condition $\mathit{Shift}(s_L, n)$ over $L_1$, such that for each graph morphism $m_1 : L_1 \to G$, there exists a graph morphism $m_0 = m_1 \circ s_L$ s.t. $m_0 \models n \Leftrightarrow m_1 \models \mathit{Shift}(s_L, n)$ (see Fig. 9).

In the refactoring example, left-hand sides of *Rule D* and *E* are isomorphic; therefore, graph morphisms in both directions exist. Since their NACs are isomorphic as well, shifting the NAC of one rule over the graph morphism produces the NAC of the other rule, because both NACs are satisfied for the same input graphs.

To support the shifting of NACs across chains of morphisms, we require the *Shift* operation to be compositional. The following lemma shows that *Shift* satisfies this requirement.

**Lemma 1 (Compositionality of Shift).** Given a NAC $n$ over $L_0$ and morphisms $s_L : L_0 \to L_1, s_{L'} : L_1 \to L_2$, the Shift operation is compositional: $\mathit{Shift}(s_{L'}, \mathit{Shift}(s_L, n)) = \mathit{Shift}(s_{L'} \circ s_L, n)$.

*Proof.* Given a graph morphism $m : L_2 \to G$, the following equivalences hold: $m \models \mathit{Shift}(s_{L'}, \mathit{Shift}(s_L, n)) \Leftrightarrow m \circ s_{L'} \models \mathit{Shift}(s_L, n) \Leftrightarrow (m \circ s_{L'}) \circ s_L \models n \Leftrightarrow m \circ (s_{L'} \circ s_L) \models n \Leftrightarrow m \models \mathit{Shift}(s_{L'} \circ s_L, n)$ □

Finally, we assume the existence of multi-pullbacks and -pushouts [38], categorical constructions generalizing the intersection and union of structures. For application of our approach to typed attribute graphs, we can use the fact that multi-pullbacks in the category of graphs exist and are well-typed, i.e., they are objects in the category of typed attribute graphs [60]. Conversely, due to co-completeness of this category [27], all colimits exist, including multi-pushouts.

**Definition 4 (Multi-pushout and multi-pullback).** Let a family of graph morphisms $(g_i : G_i \to G_t)$ with $1 \leq i \leq n$ be given. The *multi-pullback* of $g_i$ is a family of graph morphisms $(h_i : G \to G_i)$ satisfying the limit property in the following sense (see Fig. 10): For any family $(h'_i : G' \to G_i)$ with $g_i \circ h'_i = g_j \circ h'_j$, $1 \leq i \neq j \leq n$, there is a unique morphism $h_t : G' \to G$ with $h_i \circ h_t = h'_i$ for all $1 \leq i \leq n$.
In a dual manner, the colimit of a family of graph morphisms $(k_i : G \to G_i)$ is called *multi-pushout*.

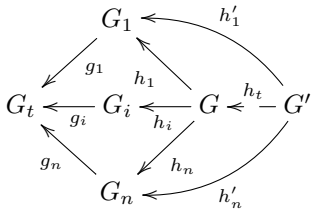


**Fig. 10.** Multi-pullback over a set of graph morphisms $g_i$ with $1 \leq i \leq n$.

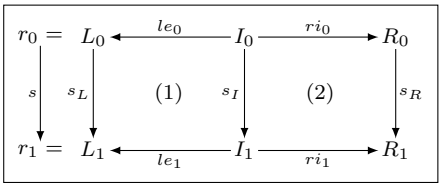


**Fig. 11.** A schematic depiction of subrule morphisms.

The following example illustrates the case where *n=2*; we focus on the interface graphs of rules, i.e., nodes and edges marked *preserve*. The interface graphs of *Rules G* and *H* overlap in nodes #53–56 and #59–62; the *preserve* edges are not part of the overlap since they differ. To build the multi-pushout of these interface graphs over their overlap, the *preserve* nodes are glued together while the separate *preserve* edges are added in: the result graph is isomorphic to the interface graph of *Rule G+H*. The multi-pullback over the embeddings between the interface graphs of *Rules G and H* to *Rule G+H* yields exactly their overlap again.

## 4. Variability-Based Model Transformation

In this section, we introduce variability-based transformation rules and show how to apply them.

### 4.1. Variability-Based Rules

We denote variability using *variability expressions*, propositional expressions over a set of *variation points*. We consider two kinds of variability expressions: the variability constraint and variability conditions. The variability constraint is used to express relationships between variation points, such as mutual exclusion or implication. Variability conditions are used to specify when specific variants expressed using variation points shall be active. The set of variation points and the variability constraint are fixed for the set of rules and not changed by transformation steps.

A *subrule* encapsulates a subset of actions on a substructure of a set of rules. If we want to identify substructures of the same rule, we talk about subrule *embeddings*. To identify common substructures of multiple rules, we talk about *subrule morphisms*.

**Definition 5 (Subrule morphism).** Given a pair of rules $r_0 = (L_0 \xleftarrow{le_0} I_0 \xrightarrow{ri_0} R_0, \mathit{NAC}_0)$ and $r_1 = (L_1 \xleftarrow{le_1} I_1 \xrightarrow{ri_1} R_1, \mathit{NAC}_1)$ (Def. 1) with injective graph morphisms $le_i, ri_i$ for $i \in \{0, 1\}$, a *subrule morphism* $s : r_0 \to r_1$, $s = (s_L, s_I, s_R)$ consists of injective morphisms $s_L : L_0 \to L_1$, $s_I : I_0 \to I_1$, and $s_R : R_0 \to R_1$ s.t., in the diagram in Fig. 11, (1) and (2) commute. In addition,

- the intersection of $s_L(L_0)$ and $le_1(I_1)$ in $L_1$ is isomorphic to $I_0$,
- the intersection of $s_R(R_0)$ and $ri_1(I_1)$ in $R_1$ is isomorphic to $I_0$,
- $L_1 \setminus (s_L(L_0) \setminus s_L(le_0(I_0)))$ is a graph, and
- for each NAC $n_0 \in \mathit{NAC}_0$, there is a NAC $\mathit{Shift}(s_L, n_0) \in \mathit{NAC}_1$.

The conditions prefaced by "in addition" ensure that a subrule performs the same actions on related elements as the original rule. The first two ensure that corresponding subrule elements have the same *delete* and *create* action in the superrule. The third one prevents the superrule rule from being applicable when the subrule is not due to dangling

edges. The fourth one ensures that the superrule cannot perform any actions being forbidden by the smaller one due to NAC violations.

**Definition 6 (Subrule embedding).** Given rules $r_0$ and $r_1$ as above. Subrule morphism $s : r_0 \to r_1$ 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 $s' \circ t_0 = t_1 \circ s$.

In the example rule shown in Fig. 6, the rule comprising all nodes and edges with no visible presence condition, i.e., nodes #65–68 and the *delete* edges, is a subrule of the entire rule. The two rules are related by a subrule embedding.

**Definition 7 (Language of variability expressions).** Given a set of variation points $V$, $\mathcal{L}_V$ is the set of all propositional expressions over $V$, called *language of variability expressions*.

The variation points in the state machine example are called *G* and *H*. The language of variability expressions includes the words *G* ∧ *H*, ¬ *G*, and *true*.

**Definition 8 (Variability constraint).** Given a language of variability expressions $\mathcal{L}_V$, a variability constraint $v$ is an element of $\mathcal{L}_V$. A total function *cfg* : $V \to \{true, false\}$ is a *variability configuration*. A variability configuration *cfg* is *valid* w.r.t. to a given variability constraint $v$ iff $v$ evaluates to *true* when each variable $vr$ in $v$ is substituted by *cfg*$(vr)$.

In the state machine example, variation points *G* and *H* are mutually exclusive since they represent the input rules of the same name. Consequently, the variability constraint is *G* $xor$ *H*, rendering the configuration *{G=true, H=true}* invalid.

**Definition 9 (Variability condition).** Given a language of variability expressions $\mathcal{L}_V$ and a variability constraint $vcs$, a *variability condition* is an element of $\mathcal{L}_V$, i.e., a propositional expression over $V$. A variability configuration *cfg* *satisfies* a variability condition $vc$ if $vc$ evaluates to *true* when each variable $vr$ in $vc$ is substituted by *cfg*$(vr)$. A variability condition is *satisfiable* if there is a valid variability configuration satisfying it. A variability condition $X$ is *stronger* than $Y$ iff $X \implies Y$.

In the example, $V = \{G, H\}$. Satisfiable variability conditions include *true*, ¬*G*, and *G* ∨ *H*; *G* ∧ *H* is not valid.

**Definition 10 (Variability-based (VB) rule).** Given $\mathcal{L}_V$, a *VB rule* $\check{r} = (r, S, v, vc)$ consists of a rule $r$, a set $S$ of subrule embeddings, a variability constraint $v$ and a function $vc : S \cup \{r\} \to \mathcal{L}_V$. Function $vc$ defines *variability conditions* for subrule embeddings s.t. $vc(id_r)$ is *true* and $\forall s \subseteq s' : vc(s') \implies vc(s)$.

For example, Figs 6 shows a VB rule in a compact representation: instead of subrule embeddings, elements are annotated. Rule $r$ is the entire rule, ignoring annotations; the variability constraint is *G* $xor$ *H*. Set $S$ and function $vc$ are derived easily by creating a subrule $r_{conj}$ for each conjunction *conj* $= \bigwedge_{i \in V} l_i$ of literals over $V$, and setting $vc(id_{r_{conj}}) =$ *conj*. Each subrule $r_{conj}$ includes the elements whose presence condition is implied by *conj*. For instance, subrule $r_{\neg G \wedge H}$ comprises elements annotated with ¬*G*, *H*, and *true*, i.e., it is isomorphic to *Rule H*. We may further include arbitrary subrules into this rule set. Doing so allows us to improve the performance of the application of VB rules. In the example, we add one additional rule: the base rule $r_{true}$ comprising all elements annotated *true*. We then have $S = \{s_{true} : r_{true} \to r,\ \ s_{\neg G \wedge H} : r_{\neg G \wedge H} \to r,\ \ s_{G \wedge \neg H} : r_{G \wedge \neg H} \to r\}$. Per Def. 5, *NAC*$_r$ is a superset of all shifted NACs of all subrules, as is the case in this example.

## 4.2. Rule merging

Given a rule set with similar rules, *rule merging* aims to find an efficient representation of these rules using a set of variability-based (VB) rules. To this end, we define a formal framework of three components as shown in Fig. 12: *clone detection* identifies *clones*, i.e., overlapping parts between rules, *clustering* assigns rules to clusters based on their clones, and *merge construction* unifies rules to create VB rules. We specify the input and output of each component and show correctness of rule merging based on these specifications. Each component may be instantiated in various ways, as long as its specification is implemented.

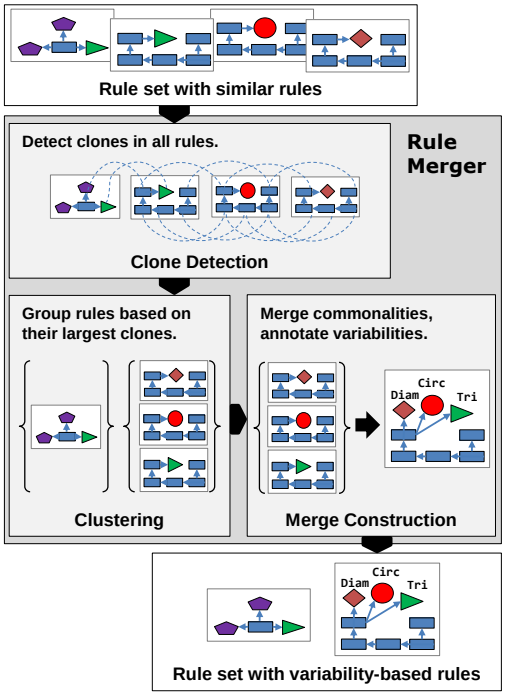


**Fig. 12.** Overview of rule merging.

### *4.2.1. 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: To cluster rules based on their similarity, we consider rules as similar if they share a large overlap. Merging overlapping portions rather than individual elements allows us to preserve the essential structural information expressed in the rules. Moreover, the performance of merged rules in terms of their execution time can be considerably improved by restricting clone detection to *connected* portions: matching connected patterns is a problem that can be handled much more efficiently than that of multiple independent patterns [70].

Formally, given a set of rules, a *clone* is a largest subrule that can be embedded into a subset of this rule set; the clone together with the embedding morphisms forms a *clone group*. To support different clone detection techniques, including heuristic ones, we do not require the set of clone groups of rule set to be complete or uniquely defined. However, the set of clone groups is always non-empty, since the empty rule can be embedded into each rule and, therefore, can potentially be a clone. To support clone detection techniques that are restricted to connected portions, we define *connected clones* based on largest connected subrules. To later establish a well-defined merge construction, we define a *compatibility* relation. Compatibility ensures that two clones never assign the same object contained in one rule to diverging objects contained in another.

**Definition 11 (Clone group).** Given a non-empty set $\mathcal{R} = \{r_i | i \in I\}$ of rules, a *clone group* $CG_{\mathcal{R}} = (r_c, \mathcal{C})$ over $\mathcal{R}$ consists of a rule $r_c$, called *clone*, and a set $\mathcal{C} = \{c_i | i \in I\}$ of subrule morphisms $c_i : r_c \to r_i$ iff there is no set $\mathcal{C}' = \{c'_i | i \in I\}$ of subrule morphisms $c'_i : r'_c \to r_i$ with a subrule morphism $s : r_c \to r'_c$ where $r'_c$ is a rule, $c_i = c'_i \circ s$ for all $i \in I$, and $s$ is not an isomorphism. If $r_c$ is connected, $CG_{\mathcal{R}}$ is called *connected* as well.
Two clone groups $CG_{\mathcal{R}} = (r_c, \{c_i | i \in I\})$ and $CG_{\mathcal{R}'} = (r'_c, \{c_j | j \in J\})$ with $\mathcal{R}' \subseteq \mathcal{R}$ and $J \subseteq I$ are *compatible* if there is a subrule morphism $s : r_c \to r'_c$ with $c_j = c'_j \circ s$ for all $j \in J$.

Table 1 shows the result of applying a clone detection technique to the classic rules in the example shown in Fig. 2. Each row denotes a clone group, comprising a set of rules and a clone present in each of these rules. We show the involved nodes for each clone; in addition, the clones incorporate edges that in each rule connect a pair of nodes involved in the clone. The rows are ordered by the size of the clone, calculated as the number of involved nodes and edges. In particular, considering rules *E* and *F*, CG2 represents objects #19-23, #25-29 and their connecting edges. CG1 incorporates nodes #24 and #30 as well as their incoming edges. Clone groups CG1 and CG2 are compatible:

| Name | Rules | Involved nodes | Size (incl. edges) |
|---|---|---|---|
| CG1 | {*E*, *F*} | {#19–24}, {#25–30} | 12 |
| CG2 | {*D*, *E*, *F*} | {#14–18}, {#19–23}, {#25–29} | 10 |
| CG3 | {*C*, *E*, *F*} | {#9–13}, {#19–21,#23–24}, {#25–27,#29–30} | 9 |
| CG4 | {*B*, *C*} | {#5–8}, {#9–12} | 8 |
| CG5 | {*A*, *B*, *C*, *D*, *E*, *F*} | {#1–4}, {#5–8}, {#9–12}, {#14–16,#18}, {#19–21,#23}, {#25–27,#29} | 7 |

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

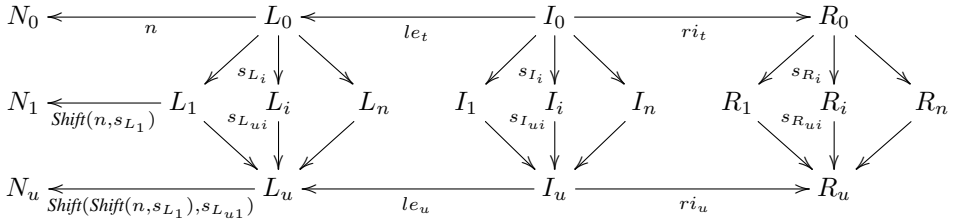


**Fig. 13.** Rule multi-pushout of subrule morphisms $s_i = (s_{L_i}, s_{I_i}, s_{R_i})$ with $1 \leq i \leq n$.

the clone of CG1 extends the one of CG2. CG2 can be reduced to the rule set of $\{E, F\}$ by discarding the subrule embedding into rule $D$. CG2 and CG3 are not compatible: their rule sets are not in subset relation. All clone groups in this example are connected.

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.

### *4.2.2. Clustering*

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

**Definition 12 (Cluster).** A *cluster* $Cl_R$ over a set $R$ of rules is a set of clone groups $CG_{R'}$ over each non-empty subset $R' \subseteq R$. Given a partition $\mathcal{P}$ of a set $\mathcal{R}$ of rules, a *cluster partition* is a set $ClPar_{\mathcal{P}}$ of clusters where for each rule set $P \in \mathcal{P}$ there is a cluster $Cl_P \in ClPar_{\mathcal{P}}$ over $P$. Each cluster $Cl_P \in ClPar_{\mathcal{P}}$ is called *a sub-cluster* of $ClPar_{\mathcal{P}}$.

In the example, we first consider a cluster over the set of rules $R$ = {*D*, *E*, *F*}. For each subset $R'$ of $R$ with two or more elements, the set of clone groups is obtained by taking those clone groups from Table 1 that refer to each rule in $R'$, and restricting the contained embeddings to $R'$. For example, for subset {*D,E*}, the cluster comprises restricted versions of CG2 and CG5, in which all mappings not referring to either *D* or *E* are discarded. For each singleton subset, there is exactly one clone group, pairing the contained rule as clone with its identity morphism as the embedding.

Given the set of rules $\mathcal{R}$ = {*A*, *B*, *C*, *D*, *E*, *F*}, a cluster partition arises over the partition {{*A*}, {*B*, *C*}, {*D*, *E*, *F*}}. The sub-cluster over {*A*} is a singleton set, in which the clone group specifies the entire rule A as clone. The sub-cluster over {*D*, *E*, *F*} is identical to the one described above, and the one over {*B*, *C*} is constructed analogously.

The output of clustering is a cluster partition over the original set of rules.

### *4.2.3. 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 the corresponding elements. To unify the rules that a specific clone is embedded into, we define an operation called *rule multi-pushout*.

**Definition 13 (Rule multi-pushout).** Let a family of subrule morphisms $(s_i : r_0 \to r_i)_{1 \leq i \leq n}$ for subrule $r_0 = (L_0 \xleftarrow{li_0} I_0 \xrightarrow{ri_0} R_0, NAC_0)$ be given. The *multi-pushout* $r_u = (L_u \xleftarrow{le_u} I_u \xrightarrow{ri_u} R_u, NAC_u)$ over $s_i$, illustrated in Fig. 13, comprises graphs $L_u, I_u, R_u$ constructed as multi-pushouts over the families $(s_{L_{ui}}), (s_{I_{ui}}), (s_{R_{ui}})$, and graph morphisms $le_u, ri_u$ obtained via the pushout property. $NAC_u$ is the union of all NACs of each $L_i$, shifting each of these NACs over $s_{L_{ui}}$ to attach them to $L_u$.

We use this operation to merge the rules contained in a cluster, based on the available information on clones. In case of several isomorphic NACs in $NAC_u$, it is enough to keep one representative and to erase all the others.

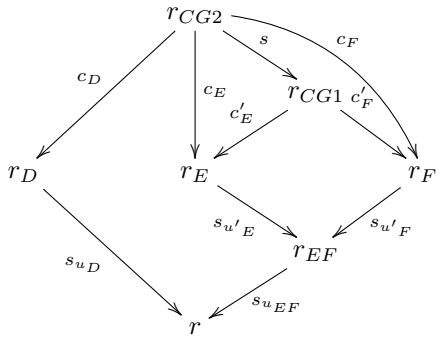


**Fig. 14.** Cluster merge by successive (multi-)pushouts.

To maintain traceability between original and new rules, we define a variation point for each original rule. The variability constraint is set over the variation points, specifying that exactly one of them is active at a time.

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. One such strategy is elaborated in Sec. 5.1.

**Definition 14 (Cluster merge).** Given a cluster partition $ClPar_{\mathcal{P}}$ over a cluster *Cl* over $\mathcal{R}$, each sub-cluster $Cl_P \in ClPar_{\mathcal{P}}$ is merged to a variability-based rule $\hat{r} = (r, S, v, vc)$ by successive multi-pullback construction over compatible clone groups in *Cl*. 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, $v = Xor_{j \in J}(v_j)$ and $vc(s_j) = v_j$. We use the notation $Merge(Cl_P)$ to indicate $\hat{r}$ and $Merge(Cl) = \{Merge(Cl_P) | Cl_P \in ClPar_{\mathcal{P}}\}$.

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. The successive construction of $r$ is illustrated in Fig. 14: first, rules *E* and *F* are merged based on CG1 to a rule $r_{EF}$. This rule is merged with *D*, based on the information in CG2, yielding rule $r$. The resulting VB rule is isomorphic to *D*+*E*+*F* in Fig. 3. In the compact representation, the presence condition of an element is the disjunction of all variation points whose corresponding subrules contain the element. The variability constraint $v$ is set to $xor(cfg(v_D), cfg(v_E), cfg(v_F))$.

In this case, the resulting VB rule has only one NAC, since the NACs of all input rules are isomorphic after shifting. The example in Fig. 6 presents an alternative case where the NACs of the original rules were not identical: in this case, the result rule contains the NACs of all input rules, being annotated with presence conditions to indicate the subrule each NAC belongs to.

As a prerequisite for showing the correctness of rule merging, we introduce an operation for *flattening* a variability-based rule, i.e., representing it by a set of classic rules. To this end, in addition to multi-pushouts (Def. 13), we need a multi-pullback construction for rules.

**Definition 15 (Rule multi-pullback).** Let a family of subrule morphisms $(s_{ri} : r_i \to r)_{1 \leq i \leq n}$ for a rule $r = (L \overset{li}{\leftarrow} I \overset{ri}{\rightarrow} R, NAC_r)$ be given. The *multi-pullback* $r_t = (L_t \overset{le_t}{\leftarrow} I_t \overset{ri_t}{\rightarrow} R_t, NAC_t)$ of $(s_i)$ comprises graphs $L_t, I_t, R_t$ constructed as multi-pullbacks over the families $(s_{L_{ri}}), (s_{I_{ri}}), (s_{R_{ri}})$ as given in Fig. 15, and graph morphisms $le_t, ri_t$ obtained via the pullback property.
The set of negative application conditions $NAC_t$ is obtained as follows:
$\forall n \in NAC_r$: if $\exists (n_i)_{1 \leq i \leq n}$ with $n_i \in NAC_i$, s.t. $n$ is isomorphic to $Shift(s_{L_{ri}}, n_i)$, and $n_i \circ s_{L_{ri}} = n_j \circ s_{L_{rj}}$ for all $1 \leq i \neq j \leq n$, there is a NAC $n_t \in NAC_t$ s.t. $n_t$ is isomorphic to $Shift(s_{L_i}, n_i)$ for all $1 \leq i \leq n$.

In a VB rule, the multi-pullback $r_t$ of the set of subrule embeddings $S$ is called *base rule*, i.e., it is the rule comprising the elements contained in all subrules.

The construction of NACs entails that a NAC is created whenever the input rules all share a completely overlapping NAC, which is the case for rules *A*–*F*. In the multi-pullback over rules *G and H*, the set of NACs is empty: some pairs of individual elements are shared between NACs, e.g. #57 and #63, but there are no completely overlapping NACs.

The following lemma shows that the multi-pullback construction is compatible to matching.

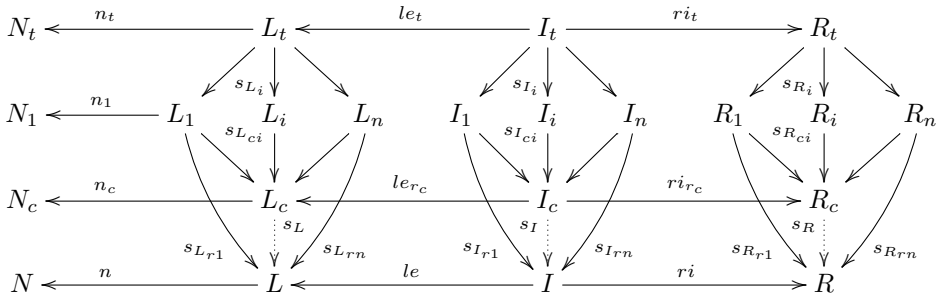


**Fig. 15.** Rule multi-pushout $(s_{ci}) = (s_{L_{ci}}, s_{I_{ci}}, s_{R_{ci}})$ of rule multi-pullback $(s_i) = (s_{L_i}, s_{I_i}, s_{R_i})$ over morphism family $(s_{ri}) = ((s_{L_{ri}}, s_{I_{ri}}, s_{R_{ri}})), 1 \leq i \leq n$.

**Lemma 2.** Let a family of subrule morphisms $(s_{ri} : r_i \to r)_{1 \leq i \leq n}$ for a rule $r$ be given together with its rule multi-pullback $r_c$, with $r = (L \xleftarrow{li} I \xrightarrow{ri} R, \mathit{NAC}_r)$, $r_i = (L_i \xleftarrow{l_{ii}} I_i \xrightarrow{r_{ii}} R_i, \mathit{NAC}_i)$, and $r_c = (L_c \leftarrow I_c \rightarrow R_c, \mathit{NAC}_c)$. Furthermore, let a match family $(m_i : L_i \to G)_{1 \leq i \leq n}$ be given where each $m_i$ satisfies $\mathit{NAC}_i$. Then, the induced morphism $m_c : L_c \to G$ satisfies $\mathit{NAC}_c$.

*Proof.* By construction, $\mathit{NAC}_c$ comprises the union of the shifted NACs $\mathit{Shift}(s_{L_{ci}}, n_i)$ of all NACs $n_i \in \mathit{NAC}_i$ of all subrules $r_i$ with $(s_i : r_i \to r)_{1 \leq i \leq |S_c|} \in S_c$. Since $m_c$ is constructed over a match family where each contained match $m_i$ satisfies $\mathit{NAC}_i$ for its rule $r_i$, the union of these NACs is satisfied by $\check{m}$ as well. □

**Definition 16 (Configuration-induced rule).** Let a VB rule $\check{r} = (r, S, v, vc)$ over $\mathcal{L}_V$ and a valid variability configuration $c$ be given. There exists a unique family of subrule embeddings $(s_{r_i} : r_i \to r)_{1 \leq i \leq |S_c|}$ with $s_{r_i} \in S$ s.t. $\forall s' \in S : s' \in (s_{r_i})_{1 \leq i \leq |S_c|}$ iff $c$ satisfies $vc(s')$. The *configuration-induced rule* $r_c$ is the multi-pushout of the multi-pullback of this family (see Fig. 15).

For example, for *Rule B+C*, configuration {*B=true*, *C=false*} yields a configuration-induced rule isomorphic to *Rule B*.

**Definition 17 (Flattening of a VB rule).** The flattening of $\check{r}$ is the set of all configuration-induced rules over all valid configurations: $\mathit{Flat}(\check{r}) = \{r_c \mid c : V \to \{\mathit{true}, \mathit{false}\} \ \wedge \ c \text{ is valid}\}$.

In the example, flattening *Rule D+E+F* yields a set of three rules being isomorphic to *Rules D, E*, and *F*.

As a proof of well-definedness, we show that merging a rule set and then flattening it produces the original set.

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

*Proof.* Given a cluster *Cl* over $\mathcal{R}$, for any partition of *Cl* we have $\mathit{Merge}(Cl) = \{\mathit{Merge}(Cl_P) | Cl_P \in \mathit{ClPar}_\mathcal{P}\}$. We show that for any sub-cluster $Cl_P \in \mathit{ClPar}_\mathcal{P}$, we have $\mathit{Flat}(\mathit{Merge}(Cl_P)) = P$ and assume $P = \{r_j | j \in J\}$. $\mathit{Merge}(Cl_P)$ yields $\hat{r} = (r, S, v, vc)$ as defined in Def. 14. Next, we consider $\mathit{Flat}(\hat{r})$. Since $v = \mathit{Xor}_{j \in J}(v_j)$, all valid configurations bind exactly one $v_j$ to *true*. We consider a fixed $j \in J$, yielding configuration $c_j$ with $c_j(v_j) = \mathit{true}$ and $c_j(v_i) = \mathit{false}$ for all $v_i \in J \setminus \{v_j\}$. $s_j \in S_P$ is in $S$ since $c$ satisfies $vc(s_j) = v_j$. Since there is exactly one subrule embedding $s_j$ for each $c_j$, no further merging of subrules is needed. The resulting subrules are the flat rules forming $P$. □

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, Thm. 1 ensures that all VB rule sets created by implementations of rule merging have the same flattening, i.e., they are semantically equivalent. Conversely, due to their syntactic variations, these rule sets may largely differ from the user perspective: if the overall representation is more compact, it might be easier to read and faster to execute. Addressing such quality goals during merging is up to concrete instantiations such as the one shown in Sec. 5.1.

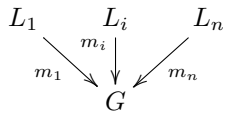


**Fig. 16.** Variability-based match family, $1 \leq i \leq n$ with $n = |S_c|$.

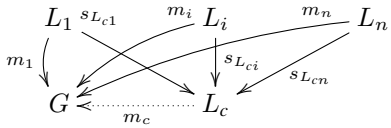


**Fig. 17.** Variability-based match, $1 \leq i \leq n$ with $n = |S_c|$.

## 4.3. Application of Variability-Based Rules

We now show how to apply variability-based rules: (1) either by flattening them to a set of classic rules and matching and then applying these rules in the classic way, or (2) directly, using a suitable variability configuration to identify a corresponding match. We then prove the equivalence of these two approaches.

### *4.3.1. Variability-based transformation through flattening*

**Definition 18 (Application of a rule set).** Given a rule set $\mathcal{R}$ and a graph $G$, the application of $\mathcal{R}$ to $G$ is the set of rule applications: $\mathit{Trans}(\mathcal{R}, G) = \{G \Rightarrow_{r,m} H\}$ with $r \in \mathcal{R}$ and a match $m : L \to G$ (Def. 2).

For example, the flattening of ***Rule G+H*** can be applied to the example state machine, yielding a set of rule applications containing exactly one element: the application of the rule isomorphic to ***Rule G*** using the match $m_G$.

### *4.3.2. Direct application of variability-based rules*

In the following, we consider the direct application of variability-based rules by finding a suitable variability-based match on-the-fly. The central task is to find variability configurations that induce a match for the left-hand side of one variant contained in the variability-based rule. If the resulting morphism of the left-hand side to graph $G$ satisfies the gluing condition for the corresponding flat rule, the rule application can take place.

**Definition 19 (Variability-based match family).** Given a variability-based rule $\check{r}$ over $\mathcal{L}_V$, a graph $G$, and a valid variability configuration $c$, there is a set of subrule embeddings $S_c \subseteq S$ s.t. $\forall r \in S : r \in S_c$ iff $c$ satisfies $vc(r)$. A *variability-based match family* is a family of matches $(m_s : L_s \to G)_{1 \leq s \leq |S_c|}$ s.t. $\forall m_i, m_j$ with $1 \leq i, j \leq |S_c|$ the following holds: $\forall x \in dom(m_i) \cap dom(m_j) : m_i(x) = m_j(x)$ (Fig. 16).

The condition ensures that matches within a family are compatible: an element contained in multiple subrules is always mapped to the same element in graph $G$. This definition entails that the identification of matches can terminate as soon as one of the subrules cannot be matched, enabling a performance benefit when only a few rules of a larger rule set are applicable: we can match the base rule $r_{\mathit{true}}$ first and only need to go on if this subrule can be matched.

In the state machine example, matching the base rule $r_{\mathit{true}}$ leads to a number of matches, including the match $m_{\mathit{base}}$ that maps nodes #65–68 plus the delete edges to node #51 with its incoming transitions and their common action. Therefore, we need to enumerate valid variability configurations to find a full VB match family. The configuration *{G=true; H=false}* induces subrules $r_{true}$ and $r_G$. A VB match family for these rules comprises $m_{\mathit{base}}$ and $m_G$, appending additional mappings for the preserve edges of rule *G*. Since no other set of matches can be found that also satisfies the NACs (Def. 1), $(m_{\mathit{base}}, m_G)$ is the only variability-based match family in this example.

**Definition 20 (Variability-based match).** Given a variability-based match family $(m_s)_{1 \leq s \leq |S_c|}$ for a variability-based rule $\check{r}$, a configuration $c$ and a graph $G$, a *variability-based match* $\check{m}$ is a pair $(m_c, c)$ where the graph morphism $m_c : L_c \to G$ is obtained by the colimit property of $L_c$ (see Fig. 17).

In the example, again considering the configuration *{G=true; H=false}*, a VB match is obtained from considering the VB match family described below Def. 19 and gluing their mappings together. This VB match has the same mappings as $m_G$.

**Definition 21 (Application of a variability-based rule).** Given a match $\check{m} = (m_c, c)$ for a variability-based rule $\check{r}$ and a graph $G$, the *application of* $\check{r}$ *at* $\check{m}$ is the classic rule application $G \Rightarrow_{r_c,m_c} H$ of the configuration-induced rule $r_c$ to $m_c$. To obtain all applications of $\check{r}$, we consider all variability-based matches: DirectTrans($\check{r}$, $G$) = $\{G \Rightarrow_{r_c,\check{m}} H \mid c$ is a valid configuration, $\check{m} = (m_c, c)$ is a variability-based match$\}$.

For example, applying of *Rule G+H* to the example state machine at the VB match with the same mappings as $m_G$ yields the same state machine described after Def. 2.

Now, we show that the set of all applications of a variability-based rule $\check{r}$ to a graph $G$ is equal to the set of classic rule applications obtained from flattening $\check{r}$ and applying these rules to $G$.

**Theorem 2 (Equivalence of rule applications).** Given a variability-based rule $\check{r}$ and a graph $G$, the following holds: *DirectTrans*($\check{r}$, $G$) = *Trans*(*Flat*($\check{r}$), $G$).

*Proof.* Since *DirectTrans*($\check{p}$, $G$) and *Trans*(*Flat*($\check{r}$), $G$) are both constructed over all valid configurations, we can consider a particular valid configuration $c : V \rightarrow \{true, false\}$ without loss of generality.

From *Trans*($Flat(\check{r}), G$), we consider the rule application $G \Rightarrow_{r_c,m} H$ of a configuration-induced rule $r_c \in Flat(\check{r})$ to a match $m_c : L_c \rightarrow G$. To obtain a family of matches $(m_s : L_s \rightarrow G)$ with $1 \leq s \leq |S_c|$, we compose $m$ with each of the embedding morphisms $(s_{L_{r_i}} : L_i \rightarrow r)$ from the construction of $r_c$. The colimit property of $L_c$ ensures that $(m_s)$ is a variability-based match family, i.e., the property required by Def. 19 holds. Per Lemma 2, $m_c$ satisfies *NAC*$_c$, as required for a match. Thus, match $m_c$ paired with configuration $c$ is a variability-based match $\check{m}$, applied with rule $r_c$ to graph $G$.

From *DirectTrans*($\check{r}$, $G$), we consider rule application $G \Rightarrow_{r_c,\check{m}} H$ of a configuration-induced rule $r_c$ to a variability-based match $\check{m}$. $r_c$ is an element of $Flat(\check{r})$. $\check{m}$ provides a match $m_c : L_c \rightarrow G$, applied with rule $r_c$ to graph $G$. □

## 5. Computation of Merge Results and Variability-Based Matches

The formal foundation introduced in Sec. 4 provides a framework that guarantees correctness of our rule merging and application mechanisms. However, it does not determine a particular computation strategy to make these mechanisms available to developers. Valid instantiations of the framework may vary heavily in the conciseness and efficiency of the produced VB rules; in particular, the trivial case of leaving all input rules as is and matching them using the classic mechanism is a valid instantiation of the framework.

In this section, we elaborate on our strategies to facilitate the computation of concise VB rules that can be matched efficiently. To this end, we provide two novel algorithms for *merge construction* and *matching of VB rules*. In addition, we elaborate on our use of existing clone detection and clustering techniques to enable merge construction.

### 5.1. Rule merging

We implemented rule merging based on state-of-the-art clone detection and clustering tools and a novel merge construction mechanisms. Our implementation features two input parameters that enable customizations with respect to specific quality goals, as we explain below.

#### *Clone Detection*

We considered the applicability of three techniques for clone detection, each of them allowing to identify *connected clones* as per Def. 11. First, we applied *gSpan*, a general-purpose graph pattern mining tool [77]. Using this tool, we experienced heap overflows even on small rule sets. Second, we re-implemented eScan [44], which terminated with *insufficient memory* errors for larger rule sets. While our implementation could be flawed, [18] reports on a similar experience with their re-implementation of eScan. Finally, we applied ConQAT [18], 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. Note that the loss of precision impacts the compactness of the produced rules, since ConQAT might fail to detect all overlaps of rules, and the undetected overlaps will remain separate during merging. The loss of precision does not, however, impair the correctness of the produced rules, since Def. 11 allows the clone detection results to be incomplete, and Theorem 2 ensures that applying the created VB rules to an input model always gives the same output models as applying each rule individually.

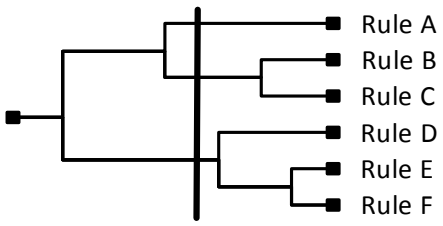


**Fig. 18.** Cluster dendrogram, as reported by clustering.

Each of these tools assumes a custom representation of its input in terms of labeled directed graphs. To make the tools applicable to model transformation rules, we developed an encoding from rules to such graphs. Our main idea was to convert each rule to one graph in the integrated representation shown in Figs. 2 and 3. Each graph element received one of the labels *create*, *delete*, *preserve*, and *forbid*. This representation ensures that cloned elements always have the same action in each rule, as required by Def. 5. NACs may overlap *partially* in the clones reported by clone detection; to ensure the correct handling of NACs as per Def. 5, we establish during merge condition that only full overlaps are considered. A detailed account of our encoding is provided in another paper [62].

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 and NACs, rather than PACs. Consequently, performance is optimized when rules are merged based on their overlap in left-hand sides and NACs. To this end, a Boolean parameter *ignoreRhs* allows restricting the rule portions considered by clone detection. When set to *true*, it only finds and reports clones for left-hand sides and NACs.

### *Clustering*

From a large variety of approaches to cluster a set of objects based on their similarity [76], 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 $\frac{12}{13} = 0.92$. It further assumes a customizable *cutting-level threshold* parameter that we describe below.

The method builds a cluster hierarchy that can be visualized using a *dendrogram*, a tree diagram arranging the input elements, as shown in Fig. 18. Tree nodes describe proximity between rule sets. The "lower" two nodes in the tree are connected, the more similar their corresponding rules are. 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. 18, and collecting the obtained subtrees. We experimented with different values for this threshold in our evaluation.

### *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 (Def. 12). To ensure a well-defined merge, merge computation refines the given cluster partition by discarding incompatible clone groups (Def. 11), 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. 14.

Fig. 19 specifies a metamodel serving as an interface between merge computation and refactoring. The class `MergeSpecification` corresponds to the overall rule set, acting as a container for a set of `MergeRules`. 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 the *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 addition, for each NAC in the

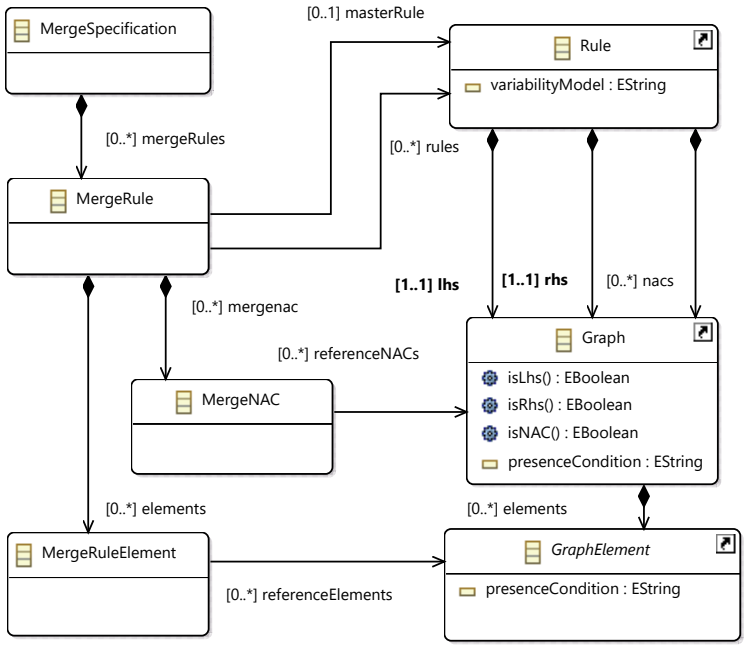


**Fig. 19.** Merge specification metamodel.

resulting rule, a `MergeNAC` is defined, referring to the NACs to be represented by it. A `MergeNAC` is only created for multiple NACs if these NACs are completely isomorphic, rendering their unification possible. Otherwise, a separate `MergeNAC` is create for each NAC. In a consistent specification, each rule element and NAC of the original rule set is referred to by exactly one `MergeRuleElement`.

Algorithm 1 sketches merge computation. The output `Merge Specification` 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* clone group – the largest available – in lines 6-8 and incrementally adding the next largest compatible ones in lines 9-13. For each clone group, we temporarily create a new `MergeRule`, integrating its contents with the result `MergeRule` in line

**Algorithm 1** Pseudocode for merge computation.

```
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:             mergeSpecification.rules.add(mergeRule)
15:             cg.removeMappings(mergeRule.rules)
16:             cg.removeAllEmpty
17:             cg.removeAll(considered)                     ▷ Done with current sub-cluster
18:     return mergeSpecification
```

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 these clone groups, we remove all empty and already considered clone groups, in lines 14-17. We repeat this process until no clone groups are left to consider.

We consider the cluster $\{D, E, F\}$ in the example. This cluster contains the clone groups CG1, CG2, CG3, and CG5; the largest one CG1 is chosen as top clone group group in line 6. In line 7, a `MergeRule` is created based on CG1, specifying the merge of rules *E* and *F*. One `MergeRuleElement` is created for each pair of clone elements and for each non-clone element, e.g., one for {#19, #25} and one for {#31}. In addition, one `MergeNAC` is created, specifying the unification of the identical NACs of *E* and *F*. In lines 9-13, 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 and NAC belongs to exactly one `MergeRuleElement` or `MergeNAC`. This process involves deleting redundant `MergeRuleElement`s and `MergeNAC`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 15–16, 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 is a straightforward implementation of Def. 14. The merge refactoring procedure is shown in Algorithm 2. The following steps are performed: all rule elements not already contained in the specified master rule are moved to this rule in line 3. in line 4, the NACs of all rules are unified, again by moving them to the master rule if they are not already contained there. In line 5, each graph element and NAC not common to all rules gets a presence condition. In lines 6 and 7, the variability model of the master rule is set and the non-master rules are removed from the overall rule set.

**Algorithm 2** Merge refactoring.

```
1: procedure MERGEREF(ms:MergeSpecification)
2:     for each merge rule mr in ms do
3:         MERGELHSRHSGRAPHS(masterRule,rules)
4:         MERGENACS(masterRule,rules)
5:         SETPRESENCECONDITIONS(elements)
6:         SETVARIABILITYMODEL(masterRule)
7:         REMOVENONMASTERRULES(rules)
```

In the example, Rule *F* has been determined as the master-rule for the cluster of $\{D, E, F\}$, since it is the largest one. As this rule already contains all elements and the NAC required for implementing the `MergeSpecification`, lines 3 and 5 have no effect. Presence conditions are created in line 5. As specified by `MergeRuleElement`s, most rule elements in *D*, *E* and *F* have corresponding elements in all other rules, rendering their presence condition to be *true*. Exceptions are objects #46, #47, and their connecting edges that now receive a non-*true* presence condition. Otherwise, elements from other rules might have been added in the specified places in the rule. Lines 6 and 7 set the variability model to mutual exclusion between variation points *D*, *E* and *F* and remove rules *D* and *E*.

## 5.2. Variability-Based Matching

We use our formal investigation of direct rule applications to provide an efficient algorithm for variability-based rule applications. The key idea is to implement the notion of variability-based matches (see Def. 20): we first match the common parts of the represented rules and then their variable parts. To ensure soundness of our NAC checking, all NACs are evaluated in a step that considers full rules rather than common portions.

Given a variability-base rule and an input model, we first match the common parts. Intuitively, the common parts of all represented rules together form the *base rule*. Formally, this notion corresponds to the multi-pullback construction in Def. 15. The base rule might have a non-empty set of NACs. Checking them now might lead to some matches of one of the represented rules not being detected later: these matches may occur when a rule binds additional elements that then cannot be bound by the base rule NAC anymore. The same applies to the gluing condition, an additional condition that needs to be fulfilled by a match (see the description after Def. 1). Therefore, at this point, we compute pre-matches rather than matches. A pre-match is a full mapping between the left-hand side of a rule and the input model that not necessarily fulfills the NACs and gluing condition. If the set of pre-matches is empty, then there also exists no match for the base rule and, therefore, no variability-based match as per Def. 20. In function FINDBASEPREMATCHES (Alg. 3), we obtain the base rule by removing all elements with non-true annotations, remove its NACs (lines 2–3), and turn the

**Algorithm 3** Pseudocode for function FINDBASEPREMATCHES.

**Input:** *model*: Input model
**Input:** *rule*: Variability-based rule
**Output:** *basePreMatches*: Base pre-matches

```
1: function FINDBASEPREMATCHES(model, rule)
2:     baseRule = rule.removeAll(elem | elem.pc ≠ true)
3:     baseRule.removeNACs()
4:     Matcher.disableCheckOfGluingCondition()
5:     return Matcher.find(model, baseRule)
```

**Algorithm 4** Pseudocode for recursive function FINDMATCHES.

**Input:** *model*: Input model
**Input:** *rule*: Variability-annotated rule
**Input:** *basePreMatches*: Pre-matches of the base rule
**Input:** *bindings*: {Variability expressions used in *rule*} → {*true*, *false*, *unbound*}
**Input:** *matches*: Accumulated variability-based matches
**Output:** *matches*: Accumulated variability-based matches

```
1: function FINDMATCHES(model, rule, basePreMatches, bindings, matches)
2:     pc_0 = bindings.select(unbound).get(0)
3:     bindings.set(pc_0, true)
4:     FINDMATCHESINNER(model, rule, basePreMatches, bindings, matches)
5:     bindings.set(pc_0, false)
6:     FINDMATCHESINNER(model, rule, basePreMatches, bindings, matches)
7:     bindings.set(pc_0, unbound)
8:     return matches
9: function FINDMATCHESINNER(model, rule, basePreMatches, bindings, matches)
10:    bindings_↓ = bindings.select(unbound).select(p | bindings.contradicts(p))
11:    bindings_→ = bindings.select(unbound).select(p | bindings.implies(p))
12:    bindings.setAll(bindings_↓ → false, bindings_→ → true)
13:    if bindings.select(unbound).isEmpty() then
14:        classicRule = rule.removeAll(elem | elem.pc ∈ bindings.select(false))
15:        classicRule = rule.removeAll(nac | nac.pc ∈ bindings.select(false))
16:        classicMatches = Matcher.find(model, classicRule, basePreMatches)
17:        matches.addAll(createVariabilityBasedMatches(classicMatches))
18:    else
19:        FINDMATCHES(model, rule, basePreMatches, bindings, matches)
20:    bindings.setAll(bindings_↓ → unbound, bindings_→ → unbound)
21:    return
```

gluing condition check off (line 4). We use the classic matching engine to match the produced base rule to the input model (line 5). The result is a set of mappings called *basePreMatches*.

Function FINDMATCHES, shown in Alg. 4, extends *basePreMatches* to find matches for the variable parts. It enumerates all valid variability configurations, derives the corresponding rules and matches them classically. FINDMATCHES receives an input model, a variability-based rule, the *basePreMatches* set, and two intermediate parameters: a data structure *bindings* that assigns each of the variability expressions used in the rule (i.e., the variability model and all used presence conditions) to one of the literals *true*, *false* or *unbound*, and a set to accumulate variability-based matches. The binding for the variability model is set to *true*, while all presence conditions are set to *unbound*. The accumulative set is initially empty. The outputs of this function is a set of variability-based matches (Def. 20).

An execution of FINDMATCHES systematically binds all presence conditions, starting on line 2 with an arbitrary one that we call $pc_0$. To enumerate all valid configurations, we first set $pc_0$ to *true* and then to *false* (lines 3-4 and 5-6). In both calls to FINDMATCHESINNER, we first consider those presence conditions that were previously unbound and now are either contradicting or implied by the current bindings. On lines 10 and 11, we compute them using a SAT solver, calling the results $bindings_{\downarrow}$ and $bindings_{\rightarrow}$ (for *false* elements and *true* elements, respectively). We update the

bindings accordingly on line 12. If all presence conditions are now bound, the problem becomes classic matching. We determine the classic rule to be matched by removing rule elements and NACs with a *false* presence condition on line 14 and 15. The classic match-finder tries to bind the rule elements contained in the derived rule, but not in the base rule. The result of this process is a set of matches, satisfying the NAC as well as the gluing condition of the considered rule. The computed matches are translated into variability-based matches, being pairs of a classic match and the current variability configuration, on lines 16–17. If some presence conditions have not been bound, we call FINDMATCHES again on line 18. On lines 7 and 19, we reset temporary bindings of variables to clean up before backtracking.

To exemplify our algorithm, we continue with the state machine example. The task is to apply the VB rule *G+H* from Fig. 6 to the Washing Machine state machine shown in Fig. 4. To this end, we derive and match the base rule, comprising elements #65–68 and the connecting *delete* edges. This results in a number of *basePreMatches*; in particular, $m_{Washing}$ binds element #50 and its incoming transitions and their common action while $m_{Unlocking}$ does the same for element #51. We arbitrarily select a presence condition $G$ and set it to *true* on line 3. Based on this choice, we derive $H$ to be *false* on lines 10-12, completing the binding of presence conditions. On lines 14 and 15, we remove all rule elements and NACs labelled $H$ to derive a rule isomorphic to *Rule G*. To find matches for this rule, we now call the classic match finder several times, using each of the *basePreMatches* as a prematch once (line 16). In this process, it shows that we cannot find a match based on the prematch $m_{Washing}$: since element #50 already has an entry action, NAC2 can never be satisfied. Instead, we find exactly one full match, extending $m_{Unlocking}$ to incorporate the two *preserve* edges, thus satisfying both NAC1 and NAC2. We pair this classic match with the current bindings to create a variability-based match. The remaining configuration {*G=false*, *H=true*} is determined analogously; however, it does not yield any additional matches.

Note that on lines 10 and 11, our algorithm relies on a SAT solver in order to enumerate valid configurations. On top of the SAT solver calls, our bookkeeping of bindings allows us to to cache evaluation results and reuse them over multiple similar SAT problems – in particular, this concerns functions *contradicts* and *implies* on lines 10 and 11. While some SAT solvers might have built-in caching features, the benefit of managing the cache ourselves is that we are more flexible w.r.t. using different SAT solvers.

Complexity of our algorithm is determined by the number of configurations which grows exponentially with the number of variation points. Of course, the configurations determine rules that in the classic approach would be matched individually. Thus, complexity of our algorithm is the same as that in classic matching. Yet, since we save matching effort by precomputing base pre-matches and then extending them, we predict that algorithm performs better than the classical one. In the next section, we study if this prediction can be confirmed.

## 6. Evaluation

We evaluated our approach by comparing it to classic model transformation and by studying the impact of the automated design decision made during merging.

To this end, we applied it to rule sets from two real-life model transformation scenarios, called FMRECOG and OCL2NGC, and one adapted from literature, called COMB. All rule sets are available as part of a publicly available benchmark set [61]. The main quality goal addressed in our evaluation is performance, for the following reasons: FMRECOG is an automatically derived rule set used in the context of model differencing [10], a task that necessitates low latency. In our communication with the developer of the OCL2NGC [5], he pointed out that the bad performance of the rule set was an obstacle to its usefulness. COMB was derived from an earlier benchmark [74]. Therefore, we optimized the two input parameters *cuttingThreshold* and *ignoreRhs*, described in Sec. 5.1, for performance.

We assessed the quality of the produced rules with respect to performance and reduction in redundancy. To quantify *performance*, we applied the rule sets ten times on all input models and measured cumulative execution time on all input models. 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 (Thm. 1). As discussed in Sec. 2, reducing redundancy in rules can benefit their maintainability.

Both real-life rule sets contained negative application conditions (NACs). Many of the rules in the FMRECOG rule set had complicated NACs, in some cases more than one per rule. Therefore, in our earlier evaluation [63], we just considered the rule set without its NACs. The NACs of OCL2NGC were relatively simple: each of the 36 rules contained a small *forbid* portion corresponding to a *create* portion of the same rule, preventing the rule from being applied twice at the same match. Our earlier implementation already contained an ad-hoc treatment for this kind of NAC being compatible with the formal foundation presented in this work. We still repeated the main experiments with our new implementation. The rules in COMB did not feature any NACs.

In what follows, we first describe each of these three scenarios in detail. Afterwards, we explain our research methodology. Finally, we present and discuss our results and address potential threats to validity.

## 6.1. Scenarios

In the first scenario, we considered a rule set taken from a product-line evolution scenario [10]. The rule set, FMRECOG, contains 54 rules for detecting applications of certain edit operations on a feature model. The rules are applied on pairs of revisions of the same feature model. To detect edit operations after they were applied – a crucial task in revision management – we need to find all matches for all rules, which is a highly performance-intensive task. To measure performance, we applied the rules in FMRECOG on nine feature models with 100 to 300 features each. The feature models were automatically generated using BeTTY [55] with parameterization profiles rendered after real-world feature models. For details, please see [10]. To create revisions, editing operations were applied randomly. Moreover, we preprocessed the rules in FMRECOG to remove instances of rule amalgamation, an advanced transformation feature that is outside the scope of this work. Still, in contrast to our earlier evaluation [63], we did not remove the 39 NACs from the rules in FMRECOG, allowing is to study the effect of the approach with its novel NAC treatment.

The second scenario, OCL2NGC, is a translator from Object Constraint Language (OCL) expressions to Nested Graph Constraints [5]. In the rule set, comprising 54 rules in total, we focused on a subset of 36 rules that are applied non-deterministically as long as one of them is applicable. This rule subset caused a significant performance bottleneck during the translation of OCL constraints. For our experiments, we refactored BRS automatically, using the automated approach, and manually, allowing to compare both approaches to merging. For the manual merging, we clustered the input rules relying on naming similarities between the rules and merged them based on symmetries that we recognized in their diagrammatic representations, a daunting and time-consuming task spanning over three days. To measure performance, we applied all rules, including their NACs, on ten OCL invariants from [5] designed for high coverage of the translation rules. The input model in each run included the actual invariant paired with the OCL standard library, yielding 1850 graph elements on average.

The third scenario is based on Varró et al.'s widely-known graph transformation benchmark *Comb Pattern* [74]. In the original benchmark, the task was to find occurrences of a small pattern – the *comb pattern* – in a large grid. The benchmark has two parameters: the size of the grid and the size of the comb. We extended the task to contain variability so the new task was to find combs of variable size *k*, where *k* can represent any integer in the range $[m_1, m_2]$. For our measurements, we considered the range $[3, 8]$, which was small enough to create the included rules manually, but large enough to expect an observable difference. We created the 6 comb pattern rules required in the classic approach. We measured performance on 10 different grids, spanning from 20x20 to 200x200 elements, which allowed us to consider a variety of input models of different sizes. We considered both sub-tasks described in the original paper: COMBNOMATCH and COMBSEVERALMATCHES. In the former, the grid is constructed to contain no occurrences of the comb pattern. In the latter, the grid is constructed to contain many such occurrences. This rule set did not contain any NACs.

## 6.2. Methods and Set-Up

In our evaluation, we investigated the following research questions:

- **RQ1:** *How do VB rules created by rule merging compare to the equivalent classical rules?*
- **RQ2:** *How do VB rules created by rule merging compare to those created manually?*
- **RQ3:** *How do the VB rules created by rule merging scale to large input models?*
- **RQ4:** *What is the impact of clone detection?*
- **RQ5:** *What is the impact of clustering?*

For RQ1, we considered all three rule sets. For RQ2, we considered the scenario where a manually created rule set was available: OCL2NGC. For RQ3, we considered the COMB scenario, as it features a procedure to increase the input model automatically (increasing the size of the input grid [74]). We measured the impact of model size on execution time until we ran out of memory. For RQ4, we randomly discarded 25%–100% of the reported clone groups. For RQ5, 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 [18] on OCL2NGC and FMRECOG, as it was the only tool

| | | *Size* | | | *Execution time (sec.)* | | | |
|---|---|---|---|---|---|---|---|---|
| *Scenario* | *Rule Set* | #Rules | #Elements | #NACs | Total | Sd | Median | Sd |
| FMRECOG | Classic | 54 | 4839 | 39 | 662.3 | 28.0 | 51.9 | 2.7 |
| | Automatic Merge | 29 | 3375 | 36 | 117.4 | 6.4 | 8.0 | 0.4 |
| OCL2NGC | Classic | 36 | 3541 | 36 | 398.5 | 23.6 | 55.0 | 4.4 |
| | Manual Merge | 10 | 1018 | 10 | 87.0 | 4.7 | 26.5 | 1.7 |
| | Automatic Merge | 18 | 2421 | 23 | 6.3 | 0.4 | 3.9 | 0.1 |
| COMB | Classic | 6 | 252 | 0 | 1.39 | 0.09 | 0.12 | 0.01 |
| NOMATCH | Automatic Merge | 1 | 62 | 0 | 0.24 | 0.09 | 0.02 | 0.01 |
| COMB | Classic | 6 | 252 | 0 | 10.4 | 0.18 | 0.83 | 0.02 |
| SEVERALMATCHES | Automatic Merge | 1 | 62 | 0 | 14.2 | 0.26 | 1.07 | 0.05 |

**Table 2.** Results for RQ1 and RQ2: Quality characteristics of the rule sets.

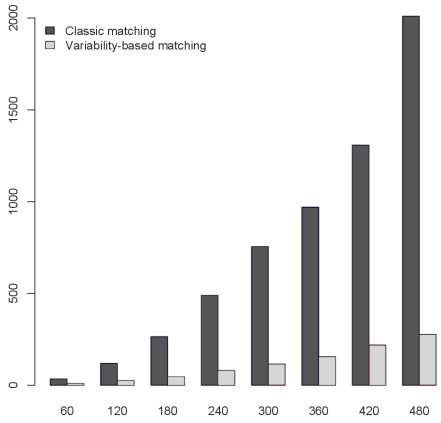


**Fig. 20.** Results for RQ3: Execution time in sec. (y) related to length of grid (x).

that scaled to these scenarios. Since ConQat only reports an approximation rather than the precise set of all clones, we applied gSpan [77] on the COMB rule set to consider all clones. The input parameters *ignoreRhs* and *cuttingThreshold* were tuned independently for each scenario by applying rule merging repeatedly until the execution time was minimized. Using this tuning approach, we determined the following final configurations: The *cuttingThreshold* was 0.57 for OCL2NGC, 0.75 for FMRECOG, and 0.8 for COMB. In each case, we set the *ignoreRhs* parameter to *true*. Moreover, the Henshin transformation engine features an optimization concerning the order of nodes considered during matching. To study the effect of our technique the FMRECOG rule set in isolation without that optimization, we deactivated it. We ran all experiments on a Windows 7 workstation (3.40 GHz processor; 8 GB of RAM).

## 6.3. 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 is denoted in terms of the total number of nodes and edges, including both left-hand and right-hand side of the involved rules.

**RQ1:** *How do VB rules created by rule merging compare to the equivalent classical rules?*
The execution time observed for OCL2NGC after the automated rule merging showed a decrease by the factor of 63. This substantial speed-up can be partly explained by the merging component of rule merging that eliminates the anti-pattern *Left-hand side not connected (LhsNC)* [70]: 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 5.6 and 5.8, respectively. When applying the COMB rule

| | *Discarded portion (d)* | | | | |
|---|---|---|---|---|---|
| *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 RQ4: Impact of considered overlap on execution time (sec.).

| | *Clustering strategy* | |
|---|---|---|
| *Scenario* | AvLinkage | Random |
| OCL2NGC | 5.8 | 80 |
| FMRECOG | 211 | 788 |

**Table 4.** Results for RQ5: Impact of clustering strategy on execution time (sec.).

set on the SEVERALMATCHES scenario, which involves an artificial input model with many possible matches [74], 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 so that it reuses results of earlier initializations. The amount of redundancy was reduced by 30% in OCL2NGC, 31% in FMRECOG, and 75% in COMB.

Considering our earlier measurements [63], the NAC treatment did not impair the performance of the OCL2NGC and FMRECOG rule sets to a notable extent. In fact, in the case of FMRECOG we observe a performance *improvement* by the factor 2 – an interesting and perhaps counter-intuitive observation that also applies for the classic OCL2NGC and FMRECOG rule sets. As we used the same machine and Eclipse configuration in both experiments, the most likely explanation for the speed-up is an update to a more recent Java version. Still, the overall performance gain between classic and variability-based rules is stable over both measurements.

**RQ2** *How do VB rules created by rule merging compare to those created manually?*
In OCL2NGC, we found a speed-up by the factor of 14. 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 (71%) was significantly greater than in those created by rule merging (30%). This finding highlights 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.

**RQ3:** *How do the VB rules created by rule merging scale to large input models?*
As shown in Fig. 20, the last input model before execution terminated with memory overflows was a 480x480 grid for both rule sets. We observed that the ratio between the execution time of applying the classic (left-hand bars) and the VB rules (right-hand bars) stayed the same in each iteration, independenty 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.

**RQ4** *What is the impact of clone detection?*
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.

**RQ5** *What is the impact of clustering?*

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 observation is related to the fact that rule merging removed the *LhsNC* antipattern. In FMRECOG, randomly clustered rules were comparable to the classic ones in terms of execution time.

### 6.4. Threats to validity and limitations

Factors affecting *external validity* include our choice of rule sets, test models and matching strategy, and the tuning of the two input parameters. While the considered rule sets represent three heterogeneous use cases, more examples are required to confirm that our approach works sufficiently well in diverse, potentially larger scenarios. To ensure that our test models were realistic, we employed the models used by their developers or described in the original benchmark. The performance of rule application depends on the chosen matching strategy, in our case, mapping this task to a constraint satisfaction problem [53]. We aim to consider the effect of alternative strategies in the future.

Tuning the two input parameters adds an initial performance overhead to applications of our rule merging mechanism. However, it is fair to assume that once the rule set has been produced, it will be applied frequently and to many different models. Since we used a variety of test models for tuning in each case, we are confident that our created VB rules will behave efficiently for other models of the same type, so that the initial performance overhead is eventually amortized. In contrast, while tempting at the first glance, we recommend to not use the same parameter values over different rules sets without an additional tuning: in our experience, the optimal values differed significantly from case to case. Moreover, the tuning of the input parameter requires the existence of test models in order to measure and compare the performance of the created rules. In a realistic application scenario, such test models are likely to be available; alternatively, one could generate them using a model generator.

With regard to *construct validity*, we focus on one aspect of maintainability, the amount of reduced redundancy. Giving a definitive answer on how to unify rules for 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.

A possible limitation of our rule application algorithm is that we resort to SAT solving in order to enumerate valid configurations during the matching process. In our evaluation rule sets, the complexity of the resulting SAT problems was so small that the used SAT solver only required a few milliseconds for solving them. Unfortunately, we can foresee that our approach does not scale up to arbitrarily large rule sets: The main issue is that the performance overhead for enumerating valid configurations grows exponentially with the number of variation points. Still, apart from the fact that such critically large rule sets have yet to appear in practice, this limitation is not specific to our approach, as the classic approach to model transformation is unable to deal with exponentially growing number of rule variants as well.

From the user perspective, another possible limitation is that our approach increases the size of individual rules, a potential impediment to readability [59]. We believe that this limitation can be mitigated by tool support. In our recent work, we proposed a tool environment that provides *editable views* on variability-based rules to developers, representing portions of a VB rule that correspond to configurations as selected by the user [65]. However, studying the usefulness of our approach and tool via a user study is left for future work.

## 7. Related Work

**Merge-refactoring in product line engineering.** Rule merging is related to a number of approaches in software product line engineering, specifically approaches that create feature-annotated representations from individual products. Nejati et al. [41] introduced the matching of Statechart models based on commonalities in their structure and behavior and applied it to merge models of telecommunication features. Ryssel et al. [54] proposed an approach for re-organizing product variants given in Matlab into annotative representations while identifying variation points. Rubin et al. [51, 52] defined a formal merge framework and instantiated it to class models and state machines, defining a number of desired qualities of the resulting model and studying how these can be best obtained. Wille et al. [75, 29] introduce a technique for the reverse engineering of variability from block diagrams based on their data-flow structures. Ziadi et al. [79] have proposed a language-independent approach for the reverse-engineering of product lines. 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 structural patterns. To our knowledge, our approach is the first that utilizes structural clone detection to identify such patterns.

**Optimization of model transformation rule application.** Our work can be considered a performance optimization for the NP-complete problem of transformation rule matching [6]. Earlier approaches in this area are mostly complementary to ours as they focus on the matching of single rules [73, 30, 36, 1].

Mészáros et al. [40] first explored the idea of considering overlapping portions in multiple rules. Their custom technique for detecting these sub-patterns, however, did not scale up to complete rule sets. Instead, they considered just two rules at a time, which enabled a moderate performance improvement of 11%. In our approach, applying clone detection and clustering techniques gives rise to an increased speed-up. The incremental graph pattern matching approach in [72] succeeds in mitigating the memory concern of Rete networks by considering shared sub-patterns. Yet, the authors report on deteriorated execution times: The index tables that map sub-patterns to partial matches grow so large that performance is impaired. Multi-query optimization has also been investigated for relational databases [56]. In the more closely related domain of graph databases, all optimization approaches we are aware of focus on single-query optimization [78].

**Clone refactoring.** Circumstances under which clones can and should be eliminated are the subject of an ongoing discussion [51]. Based on empirical observations, Kim et al. [35] 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 on stable revisions of the rule set. Specifically, clones of the second category 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 [42]. This approach has a low initial cost, but requires constant monitoring.

**Refactoring of model transformation rules.** Multiple techniques for refactoring transformation rules have been proposed. Alkhazi et all. propose a search-based approach that produces refactoring recommendations for ATL transformation programs using a multi-objective optimization algorithm [2]. Duplications in a rule set can be addressed via application of the *extract rule* refactoring, yielding an abstract rule containing the common elements. The main focus on this approach is to produce optimal results w.r.t quality metrics such as *minimal fan-in and fan-out* and *minimal number of rules*, whereas we have focused on the correctness of the result and the speed-up during rule application. Taentzer et al. present an approach to specify refactorings for graph transformation systems based on pre-defined patterns [68]. Syriani et al. devise a plan to create a design patterns catalog for model transformations [66]. Rentschler proposes a modularization technique tailored at textual model transformation languages such as ATL [48]. In [49], Rentschler et al. employ a clustering-based strategy to identify interface elements during the introduction of interfaces into legacy transformation rules. Cuadrado et al. present a reuse concept based on abstract transformation rules that can be instantiated for variants of similar meta-models [14]. The abstract transformation rules are reverse engineered from existing transformation rules. From all these complementary approaches, the one by Alkhazi et al. is the only one addressing duplications in rules.

**Variability in model transformations.** In the broader context of variability management mechanisms, variability-based rules are inspired by annotative representations of software product lines [16, 33, 51], in which optional parts are annotated in order to be removed on demand. Annotational mechanisms stand in contrast to *compositional* ones, in which modular building blocks are assembled to produce larger artifacts. To the best of our knowledge, all previous examples for variability management mechanisms in rules, e.g., [34, 58, 71], were based on the compositional paradigm. While these earlier approaches allowed specifying a product line of transformation rules, they did not provide an automated refactoring technique to create such representations. Furthermore, they did not provide any benefits for their performance. Finally, the achieved level of expressiveness was either lower than that of the proposed approach or so high that an efficient handling is prohibited. As for performance, [58, 34] report on a trade-off between better variability management and a performance overhead, the latter caused by the derivation of rules. In contrast, variability-based rules and matching improve both the compactness *and* the performance of a transformation system. As for expressiveness, [58] and [71] are based on creating refinement rules for the variable parts and assigning them to one feature (or variation point). In turn, we support propositional presence conditions over variation points. In this respect, [34] goes even further by allowing users to annotate a rule element with embedded C++ code, which, however, would produce an extremely large search space for variability-based matching.

**Rule composition.** Interestingly, despite being an annotative mechanism, variability-based model transformation bears similarities to a number of compositional mechanisms for the specification and application of rules, as is reflected in the decision to use sub-rules as a fundamental concept of our formalization. In this context, a closely related mechanism is

*rule amalgamation* [67, 7, 46], in which an interaction scheme based on a kernel rule and a set of multi-rules is defined to enable a composition of matches during the application. Similar to such an interaction scheme, a variability-based rule contains a base rule and a number of superrules. However, in contrast to interaction schemes, variability-based rules allow to restrict the applicability of superrules by variability conditions. On the other hand, amalgamation allows to apply multi-rules arbitrarily often, while super-rules of base rules are applied at most once.

Rule composition has also been considered in the work of Ghamarian and Rensink [45, 24] on integrating graph transformation with process-algebra-style compositionality. To enable reactivity, in the sense that the system at hand is modelled in relation to an environment with which it interacts continually, these works allow to decompose the involved graphs and rules into smaller units. Graphs and rules are extended with interfaces in order to compose them and support the coordination of the overall system behavior. This line of work is orthogonal to ours, as it focuses on compositionality of graphs as well as rules, whereas we assume a situation where the transformation is applied to a complete input model, i.e., graphs are not decomposed.

Another important rule composition mechanism supported by several model transformation languages is *rule refinement* [37]. In such languages, a base rule is refined by a set of sub-rules modifying it. Then, some approaches [3, 32] flatten the rules for application, i.e., compile them into simpler rules. The translational semantics in the approach proposed in RubyTL [15] is closest to ours – it applies the base rules first and then applies the refinement rules on the target model of the transformation. In contrast, our approach aims to efficiently find matches in the *source* model.

## 8. Conclusion and Future Work

In this paper, we proposed variability-based (VB) model transformation, a novel approach to improve maintainability and performance in model transformation systems. Our approach supports the unification of rules in similarity-intensive rule sets into VB rules. To improve execution performance, we harness variability explicitly during the application of the rules. We introduced rule merging as an approach for constructing VB 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. Notably, the created rules were more effective when applied to transforming large input models. In contrast to our earlier works [63, 64], the approach now comprises a powerful and flexible formal framework based on category theory; in addition, it includes a treatment of negative application conditions, one of the defining features of graph-based model transformations. 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 focus on the practical usefulness of our approach. In our recent work, we have introduced a tool environment [65] aiming to support users during the management of variability-based rules. Based on a view concept, users can configure and explore variability-based rules interactively. This way, users can inspect an individual rule relevant to their task at hand, or even compare multiple ones. To evaluate the usefulness of our tool, as well as variability-based rules as a reuse concept, a user study is the next sensible step in this direction. To make our approach more applicable to a broad variety of model transformation use-cases, another crucial feature that needs to be supported is rule amalgamation, providing a for-each operator on the rule level. Finally, beyond the application of rules, there is a variety of usage scenarios that may benefit from our compacted representations of similar rules. In the taxonomy of Thüm et al. [69], this particularly concerns the class of *family-based analyses*, in which redundant computation effort is reduced by considering the variability inside similar variants explicitly. As instances of graph transformation analysis techniques that may benefit from a family-based execution mode, we foresee model checking [47], conflict and dependency analysis [9], as well as performance prediction [31].

## References

[1] Vlad Acretoaie and Harald Störrle. Efficient Model Querying with VMQL. In *Proc. of Workshop on Combining Modelling with Search- and Example-Based Approaches*, pages 7–16. CEUR-WS.org, 2015.

[2] Bader Alkhazi, Terry Ruas, Marouane Kessentini, Manuel Wimmer, and William I Grosky. Automated refactoring of ATL model transformations: a search-based approach. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pages 295–304. ACM, 2016.

[3] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 340–355, 2014.

[4] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformation. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[5] Thorsten Arendt, Annegret Habel, Hendrik Radke, and Gabriele Taentzer. From Core OCL Invariants to Nested Graph Constraints. In *Proc. of International Conference on Graph Transformation*, pages 97–112, 2014.

[6] Mikhail J. Atallah. *Algorithms and Theory of Computation Handbook*. CRC press, 2002.

[7] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and System Modeling*, 11(2):227–250, 2012.

[8] Dominique Blouin, Alain Plantec, Pierre Dissaux, Frank Singhoff, and Jean-Philippe Diguet. Synchronization of Models of Rich Languages with Triple Graph Grammars: an Experience Report. In *Proc. of International Conference on Theory and Practice of Model Transformations*, 2014.

[9] Kristopher Born, Leen Lambers, Daniel Strüber, and Gabriele Taentzer. Granularity of conflicts and dependencies in graph transformation systems. In *Proc. of the International Conference on Graph Transformation*, pages 125–141. Springer, 2017.

[10] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. Reasoning about Product-Line Evolution using Complex Differences on Feature Models. *Journal of Automated Software Engineering*, pages 1–47, 2015.

[11] Marsha Chechik, Michalis Famelis, Rick Salay, and Daniel Strüber. Perspectives of Model Transformation Reuse. In *Proceedings of International Conference on Integrated Formal Methods*, pages 28–44. Springer, 2016.

[12] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[13] Stephen A Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[14] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Reverse Engineering of Model Transformations for Reusability. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 186–201. Springer, 2014.

[15] Jesús Sánchez Cuadrado and Jesus Garcia Molina. A Model-Based Approach to Families of Embedded Domain-Specific Languages. *IEEE Transactions on Software Engineering*, 35(6):825–840, 2009.

[16] Krzysztof Czarnecki and Michał Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. of International Conference on Generative Programming and Component Engineering*, pages 422–437. ACM, 2005.

[17] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[18] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. Model Clone Detection in Practice. In *Proc. of Workshop on Software Clones*, pages 57–64. ACM, 2010.

[19] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories. *Fundamenta Informaticae*, 74(1):31–61, 2006.

[20] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[21] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions. part 2: Embedding, critical pairs and local confluence. *Fundam. Inform.*, 118(1-2):35–63, 2012.

[22] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. M-adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation. *Mathematical Structures in Computer Science*, 24(04):240406, 2014.

[23] Michalis Famelis, Levi Lucio, Gehan Selim, Rick Salay, Marsha Chechik, James R. Cordy, Juergen Dingel, Hans Vangheluwe, and Ramesh S. Migrating Automotive Product Lines: A Case Study. In *Proc. of International Conference on Theory and Practice of Model Transformations*. Springer, 2015.

[24] Amir Hossein Ghamarian and Arend Rensink. Generalised compositionality in graph transformation. In *Proc. of the International Conference on Graph Transformation*, pages 234–248. Springer, 2012.

[25] Ulrike Golas, Annegret Habel, and Hartmut Ehrig. Multi-amalgamation of rules with application conditions in m-adhesive categories. *Mathematical Structures in Computer Science*, 24(04):240405, 2014.

[26] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[27] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *Proc. of International Conference on Graph Transformation*, pages 161–176. Springer, 2002.

[28] Frank Hermann, Susann Gottmann, Nico Nachtigall, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, and Thomas Engel. On an Automated Translation of Satellite Procedures using Triple Graph Grammars. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 50–51. Springer, 2013.

[29] Sönke Holthusen, David Wille, Christoph Legat, Simon Beddig, Ina Schaefer, and Birgit Vogel-Heuser. Family Model Mining for Function Block Diagrams in Automation Software. In *Proceedings of the International Software Product Line Conference*, pages 36–43. ACM, 2014.

[30] Ákos Horváth, Gergely Varró, and Dániel Varró. Generic Search Plans for Matching Advanced Graph Patterns. *Electronic Communications of the EASST*, 6, 2007.

[31] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, and István Ráth. Towards precise metrics for predicting graph query performance. In *Proc. of the International Conference on Automated Software Engineering*, pages 421–431. IEEE, 2013.

[32] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: A QVT-like Transformation Language. In *Proc. on Symposium on Object-Oriented Programming Systems, Languages, and Applications, Companion*, pages 719–720. ACM, 2006.

[33] Christian Kästner and Sven Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40, 2008.

[34] Amogh Kavimandan, Aniruddha Gokhale, Gabor Karsai, and Jeff Gray. Managing the Quality of Software Product Line Architectures through Reusable Model Transformations. In *Proc. of QoSA/ISARCS*, pages 13–22. ACM, 2011.

[35] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An Empirical Study of Code Clone Genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[36] Christian Krause, Matthias Tichy, and Holger Giese. Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 325–339. Springer, 2014.

[37] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reuse in Model-to-Model Transformation Languages: Are We There Yet? *Journal of Software and Systems Modeling*, pages 1–36, 2013.

[38] Mihaly Makkai and Marek Zawadowski. Duality for Simple $\omega$-Categories and Disks. *Theory and Applications of Categories*, 8(7):114–243, 2001.

[39] Martin Mann, Heinz Ekker, and Christoph Flamm. The Graph Grammar Library-a Generic Framework for Chemical Graph Rewrite Systems. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 52–53. Springer, 2013.

[40] Tamás Mészáros, Gergely Mezei, Tihamér Levendovszky, and Márk Asztalos. Manual and Automated Performance Optimization of Model Transformation Systems. *International Journal on Software Tools for Technology Transfer*, 12(3-4):231–243, 2010.

[41] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and Merging of Variant Feature Specifications. *IEEE Transactions on Software Engineering*, 38(6):1355–1375, 2012.

[42] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. Clone Management for Evolving Software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012.

[43] OMG. `http://www.omg.org/spec/UML/2.5/`. Last Accessed: January 2011.

[44] Nam H Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. Complete and Accurate Clone Detection in Graph-Based Models. In *Proc. of International Conference on Software Engineering*, pages 276–286. IEEE, 2009.

[45] Arend Rensink. Compositionality in graph transformation. In *Proc. of the International Colloquium on Automata, Languages and Programming*, pages 309–320. Springer, 2010.

[46] Arend Rensink and Jan-Hendrik Kuperus. Repotting the geraniums: On nested graph transformation rules. *ECEASST*, 18, 2009.

[47] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. of the International Conference on Graph Transformation*, pages 226–241. Springer, 2004.

[48] Andreas Rentschler. *Model Transformation Languages with Modular Information Hiding*. PhD thesis, Karlsruher Institut für Technologie, 2015.

[49] Andreas Rentschler, Dominik Werle, Qais Noorshams, Lucia Happe, and Ralf Reussner. Remodularizing Legacy Model Transformations with Automatic Clustering Techniques. In *Proc. of Workshop on the Analysis of Model Transformations*, pages 4–13. CEUR-WS.org, 2014.

[50] Elie Richa, Etienne Borde, and Laurent Pautet. Translating ATL Model Transformations to Algebraic Graph Transformations. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 183–198. Springer, 2015.

[51] Julia Rubin and Marsha Chechik. Combining Related Products into Product Lines. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 285–300. Springer, 2012.

[52] Julia Rubin and Marsha Chechik. Quality of Merge-Refactorings for Product Lines. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 83–98. Springer, 2013.

[53] Michael Rudolf. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In *Proc. of Workshop on Theory and Application of Graph Transformations*, page 238. Springer Science & Business Media, 1998.

[54] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Automatic Variation-Point Identification in Function-Block-Based Models. In *Proc. of International Conference on Generative Programming and Component Engineering*, pages 23–32. ACM, 2010.

[55] Sergio Segura, José A Galindo, David Benavides, José A Parejo, and Antonio Ruiz-Cortés. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proc. of Workshop on Variability Modelling of Software-intensive Systems*, pages 63–71, 2012.

[56] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[57] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[58] Marten Sijtema. Introducing Variability Rules in ATL for Managing Variability in MDE-based Product Lines. *Proc. of Workshop on Model Transformation with ATL*, pages 39–49, 2010.

[59] Harald Störrle. On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 518–534. Springer, 2014.

[60] Daniel Strüber. *Model-Driven Engineering in the Large: Refactoring Techniques for Models and Model Transformation Systems*. PhD thesis, Philipps-Universität Marburg, 2016.

[61] Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. Scalability of Model Transformations: Position Paper and Benchmark Set. In *Workshop on Scalable Model Driven Engineering*, pages 21–30, 2016.

[62] Daniel Strüber, Jennifer Plöger, and Vlad Acretoaie. Clone Detection for Graph-Based Model Transformation Languages. In *Proceedings of the International Conference on the Theory and Practice of Model Transformations (ICMT)*, pages 191–206. Springer, 2016.

[63] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. RuleMerger: Automatic Construction of Variability-Based Rules. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 122–140, 2016. Springer.

[64] Daniel Strüber, Julia Rubin, Marsha Chechik, and Gabriele Taentzer. A Variability-Based Approach to Reusable and Efficient Model Transformations. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 283–298. Springer, 2015.

[65] Daniel Strüber and Stefan Schulz. A Tool Environment for Managing Families of Model Transformation Rules. In *Proceedings of the International Conference on Graph Transformations (ICGT), in Memory of Hartmut Ehrig*, pages 89–101. Springer, 2016.

[66] Eugene Syriani and Jeff Gray. Challenges for Addressing Quality Factors in Model Transformation. In *Proc. of International Conference on Software Testing, Verification and Validation*, pages 929–937. IEEE, 2012.

[67] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication Based Systems*. PhD thesis, Technical University of Berlin, 1996.

[68] Gabriele Taentzer, Thorsten Arendt, Claudia Ermel, and Reiko Heckel. Towards Refactoring of Rule-Based, In-Place Model Transformation Systems. In *Proc. of Workshop on the Analysis of Model Transformations*, pages 41–46. ACM, 2012.

[69] 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.

[70] Matthias Tichy, Christian Krause, and Grischa Liebel. Detecting Performance Bad Smells for Henshin Model Transformations. In *Proc. of Workshop on the Analysis of Model Transformations*, pages 82–86. CEUR-WS.org, 2013.

[71] Salvador Trujillo, Ander Zubizarreta, Josune De Sosa, and Xabier Mendialdua. On the Refinement of Model-to-Text Transformations. In *Proc. of Jornadas de Ingenieria del Software y Bases de Datos*, pages 123–133, 2009.

[72] Gergely Varró and Frederik Deckwerth. A Rete Network Construction Algorithm for Incremental Pattern Matching. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 125–140. Springer, 2013.

[73] Gergely Varró, Katalin Friedl, and Dániel Varró. Adaptive Graph Pattern Matching for Model Transformations using Model-Sensitive Search Plans. *Electronic Notes in Theoretical Computer Science*, 152:191–205, 2006.

[74] Gergely Varró, Andy Schürr, and Daniel Varró. Benchmarking for Graph Transformation. In *Proc. of International Symposium on Visual Languages and Human-Centric Computing*, pages 79–88. IEEE, 2005.

[75] David Wille. Managing Lots of Models: the Famine Approach. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 817–819. ACM, 2014.

[76] Rui Xu, Donald Wunsch, et al. Survey of Clustering Algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[77] Xifeng Yan and Jiawei Han. gspan: Graph-Based Substructure Pattern Mining. In *Proc. of International Conference on Data Mining*, pages 721–724. IEEE, 2002.

[78] Peixiang Zhao and Jiawei Han. On Graph Query Optimization in Large Networks. *Proc. of the VLDB Endowment*, 3(1-2):340–351, 2010.

[79] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines. In *Proc. of Symposium on Applied Computing*, pages 1064–1071. ACM, 2014.