Taming Multi-Variability of Software Product Line Transformations

Daniel Strüber[1], Sven Peldzsus[1], Jan Jürjens[1,2]

{strueber,peldszus,juerjens}@uni-koblenz.de,
[1] Universität Koblenz-Landau, Koblenz, Germany,
[2] Fraunhofer Institute for Software and Systems Engineering, Dortmund, Germany

Abstract. Software product lines continuously undergo model transformations, such as refactorings, refinements, and translations. In product line transformations, the dedicated management of variability can help to control complexity and to benefit maintenance and performance. However, since no existing approach is geared for situations in which both the product line *and* the transformation specification are affected by variability, substantial maintenance and performance obstacles remain. In this paper, we introduce a methodology that addresses such *multi-variability* situations. We propose to manage variability in product lines and rule-based transformations consistently by using annotative variability mechanisms. We present a staged rule application technique for applying a variability-intensive transformation to a product line. This technique enables considerable performance benefits, as it avoids enumerating products or rules upfront. We prove the correctness of our technique and show its ability to improve performance in a software engineering scenario.

1 Introduction

Software product line engineering [1] enables systematic reuse of software artifacts through the explicit management of variability. Representing a *software product line* (SPL) in terms of functionality increments called *features*, and mapping these features to development artifacts such as domain models and code allows to generate custom-tailored products on demand, by retrieving the corresponding artifacts for a given feature selection. Companies such as Bosch, Boeing, and Philips use SPLs to deliver tailor-made products to their customers [2].

Despite these benefits, a growing amount of variability leads to combinatorial explosions of the product space and, consequently, to severe challenges. Notably, this applies to software engineering tasks such as refactorings [3], refinements [4], and evolution steps [5], which, to support systematic management, are often expressed as model transformations. When applying a given model transformation to a SPL, a key challenge is to avoid enumerating and considering all possible products individually. To this end, Salay et al. [6] have proposed an algorithm that "*lifts*" regular transformation rules to a whole product line. The algorithm transforms the SPL, represented as a variability-annotated domain model, in such way as if each product had been considered individually.

Yet, in complex transformation scenarios as increasingly found in practice [7], not only the considered models include variations: The transformation system can contain variability as well, for example, due to desired optional behavior of rules, or for rule variants arising from the sheer complexity of the involved metamodels. While a number of works [8–10] support systematic reuse to improve maintainability, *variability-based model transformation* (VB) [11, 12] also aims to improve the performance when a transformation system with many similar rules is executed. To this end, these rules are represented as a single rule with variability annotations, called *VB rule*. During rule applications, a special *VB rule application* technique [13] saves redundant effort by considering common rule parts only once. In summary, for cases where either the model or the transformation system alone contains variability, solid approaches are available.

However, a more challenging case occurs when a variability-intensive transformation is applied to an SPL. In this *multi-variability* setting, where *both* the input model and the specification of a transformation contain variability, the existing approaches fall short to deal with the resulting complexity: One can either consider all rules, so they can be "lifted" to the product line, or consider all products, so they become amenable to VB model transformation. Both approaches are undesirable, as they require enumerating an exponentially growing number of artifacts and, therefore, threaten the feasibility of the transformation.

In this paper, we introduce a methodology for SPL transformations inspired by the *uniformity principle* [14], a tenet that suggests to handle variability consistently throughout all software artifacts. We propose to capture variability of SPLs and transformations using variability-annotated domain models and rules. Model and rule elements are annotated with *presence conditions*, specifying the conditions under which the annotated elements are present. The presence conditions of model and rule elements are specified over two separate sets of features, representing SPL and rule variability. Annotated domain models and rules can be created manually using available editor support [15, 16], or automatically from existing products and rules by using merge-refactoring techniques [17, 18].

Given an SPL and a VB rule, as shown in Fig. 1, we provide a *staged* rule application technique (black arrow) for applying a VB rule to a SPL. In contrast to the state of the art (shown in gray), enumerating products or rules upfront is not required. By adopting this technique, existing tools that use transformation technology, such as refactoring engines, may benefit from improved performance.

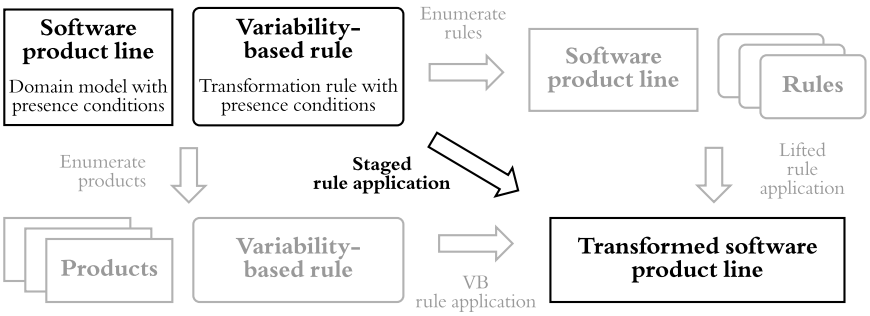


Fig. 1. Overview

Specifically, we make the following contributions:

- We introduce a staged technique for applying a VB rule to an SPL. Our technique combines core principles of VB rule applications and lifting, while avoiding their drawbacks w.r.t. enumerating all products or rules upfront.
- We formally prove correctness of this technique by showing its equivalence to the application of each "flattened" product to each "flattened" rule.
- We present an algorithm for implementing the rule application technique.
- We evaluate the usefulness of our technique by studying its performance in a substantial number of cases within a software engineering scenario.

Our work builds on the underlying framework of algebraic graph transformation (AGT) [19]. AGT is one of the standard model transformation language paradigms [20]; in addition, it has recently gained momentum as an analysis paradigm for other widespread paradigms and languages such as ATL [21]. We focus on the annotative paradigm to variability. Suitable converters to and from alternative paradigms, such as the composition-based one [22], may allow our technique to be used in other cases as well.

The rest of this paper is structured as follows: We motivate and explain our contribution using a running example in Sect. 2. Sect. 3 revisits the necessary background. Sect. 4 introduces the formalization of our new rule application technique. The algorithm and its evaluation are presented in Sects. 5 and 6, respectively. In Sect. 7 we discuss related work, before we conclude in Sect. 8.

2 Running example

In this section, we introduce SPLs and variability-based model transformation by example, and motivate and explain our contribution in the light of this example.

Software product lines. An SPL represents a collection of models that are similar, but different to each other. Fig. 2 shows a washing machine controller SPL in an annotative representation, comprising an annotated domain model

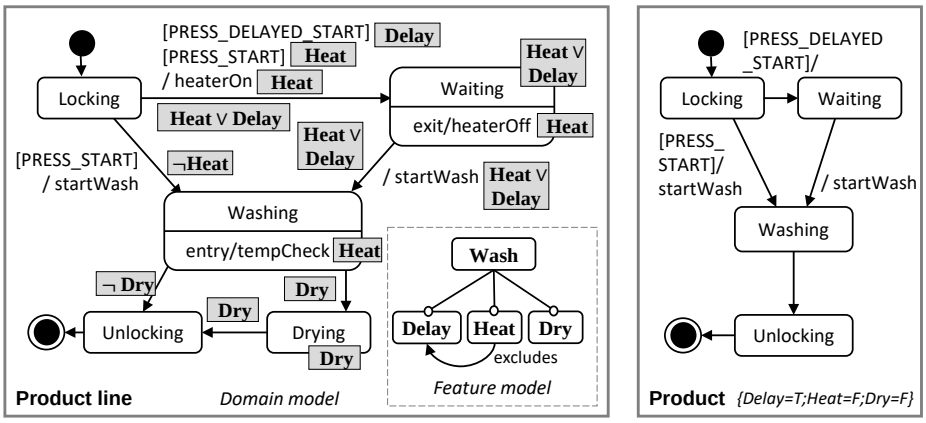


Fig. 2. Washing Machine Controller Product Line and Product (adapted from [6]).

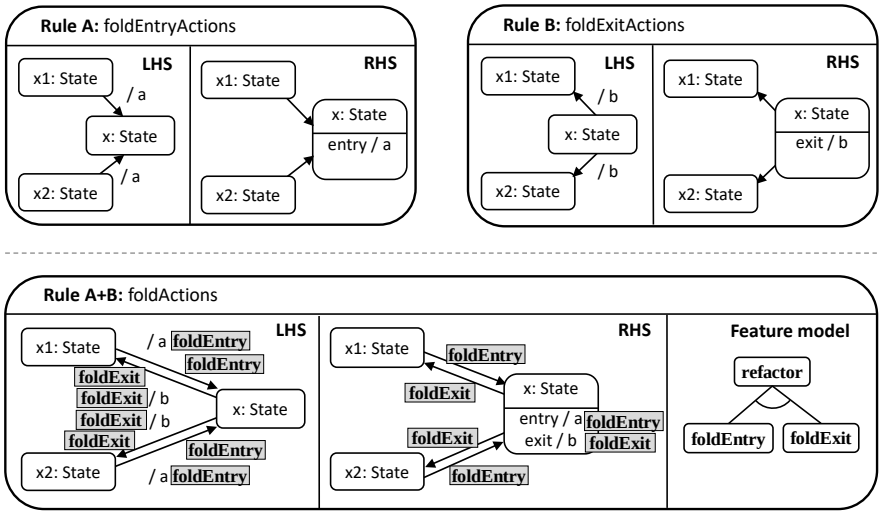


Fig. 3. Two rules and their encoding into a variability-based rule (adapted from [24]).

and a feature model. The feature model [23] specifies a root feature *Wash* with three optional children *Heat*, *Delay*, and *Dry*, where *Heat* and *Delay* are mutually exclusive. The domain model is a statechart diagram specifying the behavior of the controller SPL based on states *Locking*, *Waiting*, *Washing*, *Drying*, and *UnLocking* with transitions between them. Presence conditions, shown in gray labels, denote the condition under which an annotated element is present. These conditions are used to specify variations in the execution behavior.

Concrete products can be obtained from *configurations*, in which each optional feature is set to either *true* or *false*. A product arises by removing those elements whose presence condition evaluates to false in the given configuration. For instance, selecting *Delay* and deselecting *Heat* and *Dry* yields the product shown in the right of Fig. 2. The SPL has six configurations and products in total, since *Wash* is non-optional and *Delay* excludes *Heat*.

Variability-based (VB) model transformation. In complex model transformation scenarios, developers often create rules that are similar, but different to each other. As an example, consider two rules *foldEntryActions* and *foldExitActions* (Fig. 3), called A and B in short. These rules express a "fold" refactoring for statechart diagrams: if a state has two incoming or outgoing transitions with the same action, these actions are to be replaced by an entry or exit action of the state. The rules have a left- and a right-hand side (LHS, RHS). The LHS specifies a pattern to be matched to an input graph, and the difference between the LHS and the RHS specifies a change to be performed for each match, like the removing of transition actions, and the adding of exit and entry actions.

Rules A and B are simple; however, in a realistic transformation system, the number of required rules can grow exponentially with the number of variation points in the rules. To avoid combinatorial explosion, a set of variability-intensive rules can be encoded into a single representation using a *VB rule* [18, 12]. A VB

Table 1. Approaches for dealing with multi-variability.

	Independent combinations	
Approach	*Example*	*General case*
Naive	12	$2^{\#F_P} * 2^{\#F_r}$
VB transformation [12]	6	$2^{\#F_P}$
Lifting [6]	2	$2^{\#F_r}$
Staged application (new)	1	1

rule consist of a LHS, a RHS, a *feature model* specifying a set of interrelated features, and *presence conditions* annotating LHS and RHS elements with a condition under which they are present. Individual "flat" rules are obtained via configuration, i.e., binding each feature to either *true* or *false*. In the VB rule *A+B*, the feature model specifies a root feature *refactor* with alternative child features *foldEntry* and *foldExit*. Since exactly one child feature has to be active at one time, two possible configurations exist. The two rules arising from these configurations are isomorphic to rules A and B.

Problem statement. Model transformations such as *foldActions* are usually designed for applications to a concrete software product, represented by a single model. However, in various situations, it is desirable to extend the usage context to a *set* of models collected in an SPL. For example, during the batch refactoring of an SPL, all products should be refactored in a uniform way.

Variability is challenging for model transformation technologies. As illustrated in Table 1, products and rules need to be considered in manifold combinations. In our example, without dedicated variability support, the user needs to specify 6 products and 2 rules individually and trigger a rule application for each of the 12 combinations. A better strategy is enabled by VB model transformation: by applying the VB rule *A+B*, only 6 combinations need to be considered. Another strategy is to apply rules A and B to the SPL by *lifting* [6] them, leading to 2 combinations and the biggest improvement so far. Still, in more complex cases, all of these strategies are insufficient. Since none of them avoids an exponential growth along the number of optional SPL features ($\#F_P$) or optional rule features ($\#F_r$), the feasibility of the transformation is threatened.

Solution overview. To address this situation, we propose a *staged* rule application technique for applying a VB rule to an SPL. As shown in Fig. 4, this technique proceeds in three steps: In step 1, we consider the base rule, that is, the common portion of rules encoded in the VB rule, and match its LHS to the full domain model, temporarily ignoring its presence conditions. For example, considering rule A+B, the LHS of the base rule contains precisely states `x1`, `x2`, and `x`. A match to the domain model is indicated by dashed arrows. Using the presence conditions, we determine if the match can be mapped to any specific product. In step 2, we extend the identified base matches to identify full matches of the rules encoded in the VB rule. In the example, we would derive rules A and B; in general, to avoid fully flattening all involved rules, one can incrementally consider common subrules. An example match is denoted in terms

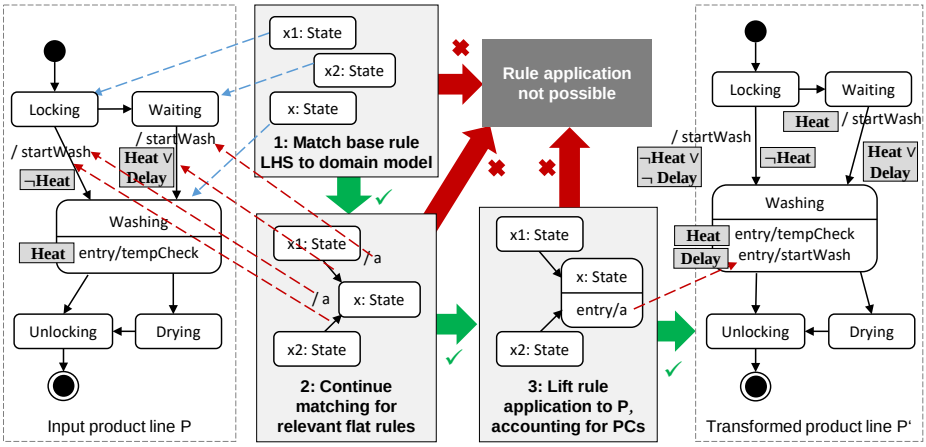


Fig. 4. Staged rule application of a VB rule to a product line.

of dashed lines for the mappings of transitions and actions. In step 3, to perform rule applications based on identified matches, we use *lifting* to apply the rule for which the match was found. Lifting transforms the domain model and its presence condition in such way as if each product was considered individually. In the example, only products for the configuration {*Delay=true; Heat=false*} are amenable to the *foldAction* refactoring. Consequently, the new entry action *startWash* has the presence condition *Delay*, and other presence conditions are adjusted accordingly. Failure to find suitable matches and to fulfill a certain condition during lifting (discussed later) allows early termination of the process.

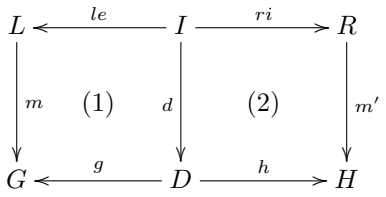
Performance-wise, the main benefit of this technique is twofold: First, using the termination criteria, we can exit the matching process early without considering specifics of products and rule variants. This is particularly beneficial in situations where none or only few rules of a larger rule set are applicable most of the time, which is typically the case, for example, in translators. Second, even if we have to enumerate some rules in step 2, we do not have to start the matching process from scratch, since we can save redundant effort by extending the available base matches. Consequently, Table 1 gives the number of independent combinations (in the sense that rule applications are started from scratch) as 1.

3 Background

We now introduce the necessary prerequisites of our methodology, starting with the double-pushout approach to algebraic graph transformation [19]. As the underlying structure, we assume the category of graphs with graph morphisms (referred to as *morphisms* from here), although all considerations are likely compatible with additional graph features such as typing and attributes.

Definition 1 (Rules and applications). *A rule* $r = (L \xleftarrow{le} I \xrightarrow{ri} R)$ *consists of graphs* L, I *and* R, *called* left-hand side, interface graph *and* right-hand side, *respectively, and two injective morphisms* le *and* ri.

Given a rule r, a graph G, and a morphism $m : L \to G$, a rule application *from G to a graph H, written $G \Rightarrow_{r,m} H$, arises from the diagram to the right, where (1) and (2) are pushouts. G, m and H are called* start graph, match, *and* result graph, *respectively.*



A rule application exists iff the match m fulfills the *gluing condition*, which, in the category of graphs boils down to the *dangling condition*: all adjacent edges of a deleted node in m's image $m[L]$ must have a preimage in L.

Product lines. Our formalization represents product lines on the semantic level by considering interrelations between the included graphs. The domain model is a "maximal" graph of which all products are sub-graphs. The presence-condition function maps sub-graphs (rather than elements, as done on the syntactic level) to terms in the boolean term algebra over features, written $T_{BOOL}(F_P)$. The set of all sub-graphs of the domain model is written $\mathcal{P}(M_P)$.

Definition 2 (Product line, configuration, product).

- *A* product line $P = (F_P, \Phi_P, M_P, f_P)$ *consists of three parts: a* feature model *that consists of a set F_P of features, and a set of* feature constraints $\Phi_P \subseteq T_{BOOL}(F_P)$, *a* domain model M_P *given as a graph, and a* set of presence conditions *expressed as a function f_P: $\mathcal{P}(M_P) \to T_{BOOL}(F_P)$.*
- *Given a set of features F, a* configuration *is a total function $c : F \to \{true, false\}$. A configuration c* satisfies *a term $t \in T_{BOOL}(F)$ if t evaluates to true when each variable v in t is substituted by $c(v)$. A configuration c is* valid *w.r.t. a set of constraints Φ if c satisfies every constraint in Φ.*
- *Given a product line $P = (F_P, \Phi_P, M_P, f_P)$, a* product *$P_c$ is derived from P under the valid configuration c if P_c is the union of all those graphs $M' \subseteq M_P$ for which $f_P(M')$ is satisfied by c: $P_c = \bigcup\{M' \subseteq M_P | c \text{ satisfies } f_P(M') \text{ and } c \text{ is valid w.r.t. } \Phi_P\}$. The* flattening *of P is the set* $\mathrm{Flat}(P)$ *of all products of P:* $\mathrm{Flat}(P) = \{P_c | P_c \text{ is a product of } P\}$.

Definition 3 (Lifted rule application). *Given a product line P, a rule r, and a match $m : L \to M_P$, a* lifted rule application $P \Rightarrow^{\uparrow}_{r,m} Q$ *is a construction that relates P to a product line Q s.t. $F_P = F_Q$, $\Phi_P = \Phi_Q$, and the set of products* $\mathrm{Flat}(Q)$ *is the same as if r was applied to each product $P_i \in \mathrm{Flat}(P)$ for which an inclusion $j : m[L] \to P_i$ from the image of m exists.*

Salay et al. [6] provide an algorithm for which it is shown that the properties required in Def. 3 apply. The algorithm extends a rule application to the domain model by a check that the match can be mapped to at least one product, and by dedicated presence condition handling during additions and deletions. A more declarative treatment is offered by Taentzer et al. [25]'s product line pushout construction, which is designed to support lifted rule application as a special case.

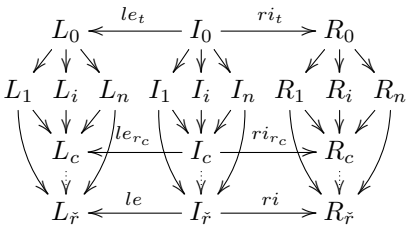
Variability-based transformation. VB rules are defined similarly to product lines, with a "maximal" rule instead of a domain model, and a notion of subrules

instead of subgraphs. A subrule is a rule that can be embedded into a larger rule injectively s.t. the actions of rule elements are preserved [12], e.g., deletions are mapped to deletions. The set of all subrules of a rule r is written $\mathcal{P}(r)$.

Definition 4 (Variability-based (VB) rule). *A VB rule* $\check{r} = (F_{\check{r}}, \Phi_{\check{r}}, r_{\check{r}}, f_{\check{r}})$ *consists of three parts: a* feature model *that consists of a set* $F_{\check{r}}$ *of features, and a set of* feature constraints $\Phi_{\check{r}} \subseteq T_{BOOL}(F_{\check{r}})$*, a* maximal rule $r_{\check{r}}$ *being a rule, and a* set of presence conditions *expressed as a function* f_P*:* $\mathcal{P}(r_{\check{r}}) \to T_{\mathrm{BOOL}}(F_{\check{r}})$.

To later consider the *base rule*, that is, a maximal subrule of multiple flat rules, we define the flattening of VB rules in terms of consecutive intersection and union constructions, expressed as multi-pullbacks and -pushouts [12]. The multi-pullback r_0 gives the base rule, over which the flat rule arises by multi-pushout.

Definition 5 (Flat rule). *Given a VB rule* $\check{r}$*, for a valid configuration* c *w.r.t.* $\Phi_{\check{r}}$*, there exists a unique set of* n *subrules* $S_c \subseteq \mathcal{P}(r_{\check{r}})$ *s.t.* $\forall s \in \mathcal{P}(r_{\check{r}}) : s \in S_c$ *iff* c *satisfies* $f_{\check{r}}(s)$*. Merging these subrules via multi-pullback and multi-pushout over* $r_{\check{r}}$ *and* r_0*, respectively, yields a rule* r_c*, called* flat rule induced by c*. The flattening of* $\check{r}$ *is the set* $\mathrm{Flat}(\check{r})$ *of all flat rules of* $\check{r}$*:*
$\mathrm{Flat}(\check{r}) = \{r_c | r_c \text{ is a flat rule of } \check{r}\}$.

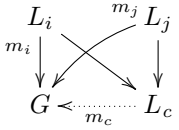


In the example, $r_{\check{r}}$ is the rule $A{+}B$, ignoring presence conditions. Given the configuration $c = \{foldEntry{=}true, foldExit{=}false\}$, the multi-pullback over each subrule whose presence condition satisfies c yields as the base rule r_0 precisely the part of rule $A + B$ without presence conditions (i.e., only the states). The resulting flat rule r_c is isomorphic to rule A.

As a prerequisite for achieving efficiency during staged application, we revisit VB rule application. The key idea is that matches of a flat rule are composed from matches of all of its subrules. By considering the subrules during matching, we can reuse matches over several rules and identify early-exit opportunities.

Definition 6 (VB match family, VB match, VB rule application).

- *Given a variability-based rule* $\check{r}$*, a graph* G*, and a valid configuration* c*, there exists a unique set of subrules* $S_c \subseteq r_{\check{r}}$ *s.t.* $\forall s \in \mathcal{P}(r_{\check{r}}) : s \in S_c$ *iff* c *satisfies* $f_{\check{r}}(s)$*. A* variability-based match family *is a family of morphisms* $(m_s : L_s \to G)_{1 \le s \le |S_c|}$ *s.t.* $\forall m_i, m_j$ *with* $1 \le i, j \le |S_c|$ *the following compatability condition holds:* $\forall x \in dom(m_i) \cap dom(m_j) : m_i(x) = m_j(x)$.
- *Given a variability-based match family* (m_s) *for* $\check{r}$*, G, and c, a* variability-based match $\check{m}$ *is a pair* (m_c, c) *where the morphism* $m_c : L_c \to G$ *is obtained by the colimit property of* L_c*. If* m_c *is a match,* $\check{m}$ *is called a* variability-based match.



- *Given a variability-based match* $\check{m} = (m_c, c)$ *for* $\check{r}$ *and* G*, the* application of $\check{r}$ at $\check{m}$ *is the rule application* $G \Rightarrow_{r_c, m_c} H$ *of the flat rule* r_c *to* m_c.

In the example, a VB match family is obtained: Step 1 collects matches of the LHS L_0. Step 2 reuses these matches to match the flat rules: according to the compatibility condition, we may extend the matches rather than start from scratch. The set of VB rule applications for a rule $\check{r}$ to a model G is equivalent to the set of rule applications of all flat rules in Flat($\check{r}$) to G ([12], Th. 2).

4 Multi-Variability of Product Line Transformations

A variability-based rule represents a set of similar transformation rules, while a product line represents a set of similar models. We consider the application of a variability-based rule to a product line from a formal perspective. Our idea is to combine two principles of *maximality*, which, up to now, were considered in isolation: First, by applying a rule to a "maximum" of all products, the rule can be lifted efficiently to a product line (Def. 3). Second, by reusing matches of a maximal subrule, several rules can be applied efficiently to a single model (Def. 6).

We study three strategies for applying a variability-based rule $\check{r}$ to a product line P; the third one leads to the notion of *staged rule application* as introduced in Sect. 2. First, we consider the naive case of flattening $\check{r}$ and P and applying each rule to each product. Second, we take the two maximality principles into account to avoid the flattening of $\check{r}$. Third, we use additional aspects from the first principle to avoid the flattening of P as well. We show that all strategies are equivalent in the sense that they change all of P's products in the same way.

4.1 Fully flattened application

Definition 7 (Fully flattened application). *Given the flattening of a product line P and the flattening of a rule family $\check{r}$, the set of fully-flattened rule applications $Trans_{FF}(P, \check{r})$ arises from applying each rule to each product:*

$$Trans_{FF}(P, \check{r}) = \{P_i \Rightarrow_{r_c, m_c} Q_i | P_i \in Flat(P), r_c \in Flat(\check{r}), match\, m_c : L_c \to P_i\}$$

In the example, there are two rules and six products; however, only for two products—the ones arising from configurations with *Delay=true* and *Heat=false*—a match, and, therefore, a rule application exists, as we saw in the earlier description of the example. $Trans_{FF}(P, \check{r})$ comprises the resulting two rule applications.

4.2 Partially flattened application

We now consider a strategy that aims to avoid unflattening the variability-based rule $\check{r}$. We use the fact that the rules in $\check{r}$ generally share a maximal, possibly empty sub-rule r_0 that can be embedded into all rules in $\check{r}$. Moreover, we exploit the fact that each product has an inclusion into the domain model.

The key idea is as follows: each match of a flat rule to a product includes a match of r_0 into the domain model M_P. Absence of such a match implies that none of the rules in $\check{r}$ has a match, allowing us to stop without considering any flat rule in its entirety. Such exit point is particularly beneficial if the VB rule represents a subset of a larger rule set in which only a few rules can be matched

at one time. Conversely, if a match for r_0 exists, a rule application arises if the match can be "rerouted" onto one of the products P_i. In this case, we consider the flat rules, saving redundant matching effort by reusing the matches of r_0.

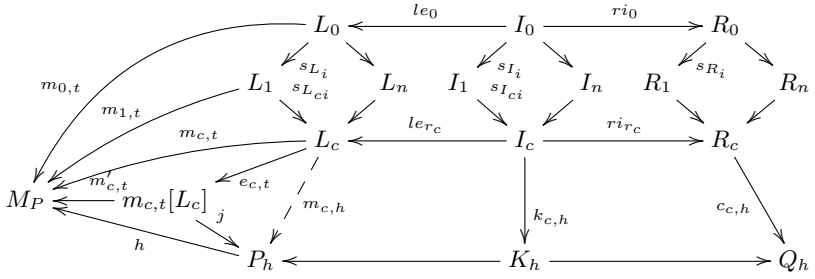
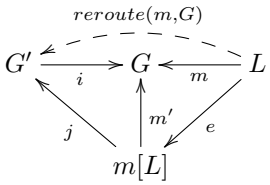


Fig. 5. Partially flattened rule application.

To reuse matches to the domain model for the products, we introduce the rerouting of a morphism from its codomain onto another graph G'. We omit naming the codomain and G' explicitly where they are clear from the context.

Definition 8 (Rerouted morphism). *Let an inclusion $i : G' \to G$, a morphism $m : L \to G$ with an epi-mono-factorization (e, m'), and a morphism $j : m[L] \to G'$ be given, s.t. $m' = i \circ j$. The* rerouted morphism *$reroute(m, G') : L \to G'$ arises by composition: $reroute(m, G') = j \circ e$*



Definition 9 (Rerouted variability-based match). *Given a graph G, a variability-based rule $\check{r}$ with a variability-based match $\check{m} = (m_c, c)$ (Def. 6), and an inclusion $i : G' \to G$. If the epi-mono-factorization of m_c and a suitable morphism j exists, a rerouted morphism onto G' arises (Def. 8). Pairing this morphism with the configuration c induces the* rerouted variability-based match *of $\check{m}_c$ onto G': $reroute(\check{m}, G') = (reroute(m_c, G'), c)$*

In Fig. 5, $m_{c,h}$ is the morphism obtained by rerouting a match $m_{c,t}$ from the domain model M_p to product P_h. For example, if $m_{c,t}$ is the match indicated in steps 1 and 2 of Fig. 4, the morphism j and, consequently, $m_{c,h}$ exists only for products in which all images of the mappings exist as well, e.g., the product shown in the right of Fig. 2. Note that $m_{c,t}$ is a variability-based match to M_P: In an earlier explanation, we saw that the family $(m_{i,t})$ forms a variability-based match family. Therefore, per Def. 9, pairing $m_{c,h}$ with the configuration c induces a variability-based match to P_h, which can be used as follows.

Variability-based rule application (Def. 6) allows us to save matching effort by considering shared parts of rules to a graph only once. The following definition allows us to lift this insight from graphs onto product lines. We show that the sets of partially and fully flattened rule applications are equivalent.

Definition 10 (Partially flattened application). *Given a variability-based rule $\check{r}$ and a product line P, the set of* partially flattened rule applications *$Trans_{PF}(P, \check{r})$ is obtained by rerouting all variability-based matches from the domain model M_P to products in P and collecting all resulting rule applications:*

$$Trans_{PF}(P, \check{r}) = \{P_i \Rightarrow_{\check{r},\check{m}'} Q_i \mid \check{m} = (m_c, c) \text{ is a VB match of } \check{r} \text{ to } M_P, \\ P_i \in Flat(P), \check{m}' = (reroute(m_c, P_i), c) \text{ is a VB match}\}$$

Theorem 1 (Equivalence of fully and partially flattened rule applications). *Given a product line P and a variability-based rule $\check{r}$, $Trans_{FF}(P, \check{r}) = Trans_{PF}(P, \check{r})$*

Proof idea.[1] For every fully flattened (FF) rule application, we can find a corresponding partially flattened (PF) one, and vice versa: Given a FF rule application at a match m', we compose m' with the product inclusion into the domain model M_P to obtain a match m_c into M_P. Per Thm. 2 in [12], m_c induces a VB match and rule application. From a diagram chase, we see that m' is the morphism arising from rerouting m_c onto the product P_i. Consequently, the rule application is PF. Conversely, a PF variability-based rule application induces a corresponding FF rule application by its definition.

4.3 Staged application

The final strategy we consider, staged application, aims to avoid unflattening the products as well. This can be achieved by employing lifting (Def. 3): Lifting takes a single rule and applies it to a domain model and its presence conditions in such a way as if the rule had been applied to each product individually. The considered rule in our case is a flat rule with a match to the domain model.

Note that we cannot compare the set of staged applications directly to the set of flattened applications, since it does not live on the product level. We can, however, compare the obtained sets of products from both sets of applications, which happens to be the same, thus showing the correctness of our approach.

Definition 11 (Staged application). *Given a variability-based rule $\check{r}$ and a product line P, the set of* staged applications *$Trans_{St}(P, \check{r})$ is the set of lifted rule applications obtained from VB matches to the domain model M_P:*

$$Trans_{St}(P, \check{r}) = \{P \Rightarrow^{\uparrow}_{r_c,m_c} Q \mid \check{m} = (m_c, c) \text{ is a VB match of } \check{r} \text{ to } M_P\}$$

Corollary 1 (Equivalence of staged and partially flattened rule applications). *Given a product line P and a variability-based rule $\check{r}$, the sets of products obtained from $Trans_{St}(P, \check{r})$ and $Trans_{PF}(P, \check{r})$ are isomorphic.*

Proof. Since both sets are defined over the same set of matches of flat rules, the proof follows straight from the definition of lifting.

[1] A full proof is provided in the appendix.

5 Algorithm

We present an algorithm for implementing the staged application of a VB rule $\check{r}$ to a product line P. Following the overview in Sect. 2 and the treatment in Sect. 4, the main idea is to proceed in three steps: First, we match the base rule of $\check{r}$ to the domain model, ignoring presence conditions. Second, we consider individual rules as far as necessary to obtain matches to the domain model. Third, based on the matches, we perform the actual rule application by using the lifting algorithm from [6] in a black-box manner.

Algorithm 1: Staged application.

Input : Product line P, VB rule $\check{r}$
Output: Transformed product line P

```
1  BMatches := findMatches(Model_P, r_0);
2  foreach m ∈ BMatches do
3  |  Φ_pc := ⋀ { pc ∈ pcs_pre };
4  |  if Φ_P ∧ Φ_pc is SAT then
5  |  |  foreach c ∈ configs(ř) do
6  |  |  |  flatRule := r_ř.removeAll(e |
   |  |  |    c ⊭ pc_e );
7  |  |  |  Matches := findMatches(
   |  |  |    Model_P, flatRule, m);
8  |  |  |  lift(P, flatRule, Matches);
9  |  |  end
10 |  end
11 end
```

Algorithm 1 shows the computation in more detail. In line 1, $\check{r}$'s base rule r_0 is matched to the domain model $Model_P$, leading to a set of base matches. If this set is empty, we have reached the first exit criterion and can stop directly. Otherwise, given a match m, in line 2, we check if at least one product P_i exists that m can be rerouted onto (Def. 8). To this end, in lines 3–4, we use a SAT solver to check if there is a valid configuration of P's feature model for which all presence conditions of matched elements evaluate to *true*. In this case, we iterate over the valid configurations of $\check{r}$ in line 5 (we may proceed more fine-grainedly by using partial configurations; this optimization is omitted for simplicity). In line 6, a flat rule is obtained by removing all elements from the rule whose presence condition evaluates to *false*. We match this rule to the domain model in line 7; to save redundant effort, we restrict the search to matches that extend the current base match. Absence of such a match is the second stopping criterion. Otherwise, we feed the flat rule and the set of matches to lifting in line 8. Handling dangling conditions is left to lifting; in the positive case, P is transformed afterwards.

For illustration, consider the base match $m_1 = \{$*Looking, Waiting, Washing*$\}$ from Fig. 4. First we calculate Φ_{pc}. As none of the states in the domain model has a presence condition, Φ_{pc} is set to *true* and is identified as satisfiable. Two valid configurations exist, $c_1 = \{$*foldEntry=true,foldExit=false*$\}$ and $c_2 = \{$*foldEntry=false,foldExit=true*$\}$. Considering c_1, the presence condition *foldExit* evaluates to false; removing the corresponding elements yield a rule isomorphic to Rule A in Fig. 3. Match m_1 is now extended using this rule, leading to a match as shown in step 2 of Fig. 4. and then lifted, as discussed in the earlier explanation of the example. Step 2 is repeated for configuration c_2; yet, as no suitable match in c_2 exists, the shown transformation is the only possible one.

This algorithm benefits from the correctness results shown in Sect. 4. Specifically, it computes staged rule applications as per Def. 11: A configuration c is

Table 2. Subject rule set.

Category	#Rules	#VBRules
Create/Set	274	171
Delete/Unset	164	121
Change/Move	966	212
Total	1404	504

Table 3. Subject product lines.

SPL	#Elements	#Products
1: InCar	116	54
2: E2E	130	94
3: JSSE	24,077	64
4: Notepad	252	512
5: Mobile	4,069	3,072
6: Lampiro	29,045	5,892

determined in line 5, and values for match m_c are collected in the set *Matches*. Via Corollary 1 and Theorem 1, the effect of the rule application to the products is the same as if each product had been considered individually.

In terms of performance, two limiting factors are the use of a graph matcher and a SAT solver; both of them perform an NP-complete task. Still, we expect practical improvements from our strategy of reusing shared portions of the involved rules and graphs, and from the availability of efficient SAT solvers that scale up to millions of variables [26]. This hypothesis is studied in Sect. 6.

6 Evaluation

To evaluate our technique, we implemented it for Henshin [27, 28], a graph-based model transformation language, and applied it to a transformation scenario with product lines and transformation variability. The goal of our evaluation was to study if our technique indeed produces the expected performance benefits.

Setup. The transformation is concerned with the detection of applied editing operations during model differencing [29]. This setting is particularly interesting for a performance evaluation: Since differencing is a routine software development task, low latency of the used tools is a prerequisite for developer effectiveness. The rule set, called `UmlRecog`, is tailored to the detection of UML edit operations. Each rule detects a specific edit operation, such as "move method to superclass", based on a pair of model versions and a low-level difference trace. `UmlRecog` comprises 1404 rules, which, as shown in Table 6, fall in three main categories: *Create/Set*, *Change/Move*, and *Delete/Unset*. To study the effect of our technique on performance, an encoding of the rules into VB rules was required. We obtained this encoding using RuleMerger [18], a tool for generating VB rules from classic ones based on clustering and clone detection [30]. We obtained 504 VB rules; each of them representing between 1 and 71 classic rules. `UmlRecog` is publicly available as part of a benchmark transformation set [31].

We applied this transformation to the 6 UML-based product lines specified in Table 6. The product lines came from diverse sources and include manually designed ones (1–2), and reverse-engineered ones from open-source projects (3–6). Each product line was available as an UML model annotated with presence conditions over a feature model. To produce the model version pairs used by `UmlRecog`, we automatically simulated development steps by nondeterministically applying rules from a set of edit rules to the product lines, using the lifting algorithm to account for presence conditions during the simulated editing step.

	Create/Set			Delete/Unset			Change/Move			TOTAL		
	lift	stage	factor	lift	stage	factor	lift	stage	factor	lift	stage	factor
InCar	2.13	0.52	**4.1**	0.23	0.12	**1.9**	7.28	0.86	**8.5**	9.66	1.49	**6.5**
E2E	1.99	0.82	**2.4**	0.35	0.32	**1.1**	7.28	0.95	**7.7**	9.62	2.12	**4.5**
JSSE	2.00	0.51	**3.9**	0.24	0.16	**1.5**	8.40	3.08	**2.7**	10.61	3.79	**2.8**
Notepad	2.05	0.66	**3.1**	0.26	0.14	**1.9**	7.01	1.64	**4.3**	9.38	2.47	**3.8**
Mobile	2.00	0.55	**3.7**	0.24	0.13	**1.9**	8.28	1.62	**5.1**	10.55	2.26	**4.7**
Lampiro	2.05	0.64	**3.2**	0.26	0.15	**1.7**	8.25	2.58	**3.2**	10.55	3.29	**3.2**

Table 4. Execution times (in seconds) of the lifting and the staged approach.

As baseline for comparison, we considered the lifted application of each rule in `UmlRecog`. An alternative baseline of applying VB rules to the flattened set of products was not considered: The SPL variability in our setting is much greater than the rule variability, which implies a high performance penalty when enumerating products. Since we currently do not support advanced transformation features, e.g., negative application conditions and amalgamation, we used variants of the flat and the VB rules without these concepts. We used a Ubuntu 17.04 system (Oracle JDK 1.8, Intel Core i5-6200U, 8GB RAM) for all experiments.

Results. Tables 4 gives an overview of the results of our experiments. The total execution times for our technique were between 1.5 and 3.3 seconds, compared to 9.4 and 10.6 seconds for lifting, yielding a speedup by factors between 2.8 and 6.5. For both techniques, all execution times are in the same order of magnitude across product lines. A possible explanation is that the amount of applicable rules was small: if the vast majority of rules can be discarded early in the matching process, the execution time is constant with the number of rules.

The greatest speedups were observed for the *Change/Move* category, in which rule variability was the greatest as well, indicated by the ratio between rules and VB rules in Table 6. This observation is in line with our rationale of reusing shared matches between rules. Regarding the number of products, a trend regarding better scalability is not apparent, thus demonstrating that lifting is sufficient for controlling product-line variability. Still, based on the overall results, the hypothesis that our technique improves performance in situations with significant product-line and transformation variability can be confirmed.

Threats to Validity. Regarding external validity, we only considered a limited set of scenarios, based on six product lines and one large-scale transformation. We aim to apply our technique to a broader class of cases in the future. The version pairs were obtained in a synthetic process, arguably one that produces pessimistic cases. Our treatment so far is also limited to a particular transformation paradigm, AGT, and one variability paradigm, the annotative one. Still, AGT and annotative variability are the underlying paradigms of many state-of-the-art tools. Finally, we did not consider the advanced AGT concepts of negative application conditions and amalgamation in our evaluation; extending our technique accordingly is left as future work.

7 Related Work

During an SPL's lifecycle, not only the domain model, but also the feature model evolves [32, 33]. To support the combined transformation of domain and feature models, Taentzer et al. [25] propose a unifying formal framework which generalizes Salay et al.'s notion of lifting [6], yet in a different direction than us: focusing on combined changes, this approach is not geared for internal variability of rules; similar rules are considered separately. Both works could be combined using a rule concept with separate feature models for rule and SPL variability.

Beyond transformations of SPLs, transformations have been used to *implement* SPLs. Feature-oriented development [34] supports the implementation of features as additive changes to a base product. Delta-oriented programming [35] adds flexibility to this approach: changes are specified using *deltas* that support deletions and modifications as well. Impact analysis in an evolving SPL can be performed by transforming deltas using higher-order deltas that encapsulate certain evolution operators [5]. For increased flexibility regarding inter-product reuse, deltas can be combined with traits [36]. Sijtema [8] introduced the concept of variability rules to develop SPLs using ATL. Conversely, SPL techniques have been applied to certain problems in transformation development. Xiao et al. [37] et al. propose to capture variability in the backwards propagation of bidirectional transformations by turning the left-hand-side model into a SPL. Hussein et al. [10] present a notion of rule templates for generating groups of similar rules based on a data provenance model. These works address only one dimension of variability, either of a SPL or a transformation system.

In the domain of graph transformation reuse, rule refinement [9] and amalgamation [38] focus on reuse at the rule level; graph variability is not in their scope. Rensink and Ghamarian propose a solution for rule and graph decomposition based a certain accommodation condition, under which the effect of the original rule application is preserved [39, 40]. In our approach, by matching against the full domain model rather than decomposing it, we trade off compositionality for the benefit of imposing fewer restrictions on graphs and rules.

8 Conclusion and Future Work

We propose a methodology for software product line transformations in which not only the input product line, but also the transformation system contains variability. At the heart of our methodology a staged rule application technique exploits reuse potential with regard to shared portions of the involved products and rules. We showed the correctness of our technique and demonstrated its benefit by applying it to a practical software engineering task.

In the future, we aim to explore further variability dimensions, e.g., metamodel variability as considered in [41], and to extend our work to advanced transformation features, such as application conditions. We aim to address additional variability mechanisms and to perform a broader evaluation.

Acknowledgement. We thank Rick Salay and the anonymous reviewers for their constructive feedback. This work was supported by the Deutsche Forschungsgemeinschaft (DFG), project *SecVolution@Run-time*, no. 221328183.

References

1. K. Pohl, F. Guenter Boeckle, and van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
2. S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2016.
3. S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake, "Variant-preserving refactoring in feature-oriented software product lines," in *VaMoS*, 2012, pp. 73–81.
4. P. Borba, L. Teixeira, and R. Gheyi, "A theory of software product line refinement," *Theoretical Computer Science*, vol. 455, pp. 2–30, 2012.
5. S. Lity, M. Kowal, and I. Schaefer, "Higher-order delta modeling for software product line evolution," in *FOSD*, 2016, pp. 39–48.
6. R. Salay, M. Famelis, J. Rubin, A. D. Sandro, and M. Chechik, "Lifting Model Transformations to Product Lines," in *ICSE*, 2014, pp. 117–128.
7. D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi *et al.*, "A research roadmap towards achieving scalability in model driven engineering," in *BigMDE*. ACM, 2013, p. 2.
8. M. Sijtema, "Introducing variability rules in ATL for managing variability in MDE-based product lines," *MtATL*, vol. 10, pp. 39–49, 2010.
9. A. Anjorin, K. Saller, M. Lochau, and A. Schürr, "Modularizing Triple Graph Grammars Using Rule Refinement," in *FASE*, 2014, pp. 340–355.
10. J. Hussein, L. Moreau *et al.*, "A template-based graph transformation system for the PROV data model," in *GCM*, 2016.
11. D. Strüber, "Model-Driven Engineering in the Large: Refactoring Techniques for Models and Model Transformation Systems," Ph.D. dissertation, Philipps-Universität Marburg, 2016.
12. D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger, "Variability-based model transformation: formal foundation and application," *Formal Aspects of Computing*, pp. 1–30, Nov 2017.
13. D. Strüber, J. Rubin, M. Chechik, and G. Taentzer, "A Variability-Based Approach to Reusable and Efficient Model Transformations," in *FASE*. Springer, 2015, pp. 283–298.
14. C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, "Language-independent safe decomposition of legacy applications into features," *Tech. Rep, School of Computer Science, University of Magdeburg, Germany*, vol. 2, 2008.
15. A. Di Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik, "MMINT: A graphical tool for interactive model management," in *P&D@ MoDELS*, 2015, pp. 16–19.
16. D. Strüber and S. Schulz, "A Tool Environment for Managing Families of Model Transformation Rules," in *ICGT*. Springer, 2016, pp. 89–101.
17. J. Rubin and M. Chechik, "Combining related products into product lines," in *FASE*, vol. 12. Springer, 2012, pp. 285–300.
18. D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger, "RuleMerger: Automatic Construction of Variability-Based Rules," in *FASE*. Springer, 2016, pp. 122–140.
19. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.
20. K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

21. E. Richa, E. Borde, and L. Pautet, "Translation of ATL to AGT and application to a code generator for Simulink," *SoSyM*, pp. 1–24, 2017.
22. C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *ICSE*, 2008, pp. 311–320.
23. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.
24. M. Chechik, M. Famelis, R. Salay, and D. Strüber, "Perspectives of Model Transformation Reuse," in *iFM*. Springer, 2016, pp. 28–44.
25. G. Taentzer, R. Salay, D. Strüber, and M. Chechik, "Transformations of software product lines: A generalizing framework based on category theory," in *MODELS*. IEEE, 2017, pp. 101–111.
26. C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Foundations of Artificial Intelligence*, vol. 3, pp. 89–134, 2008.
27. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformation," in *MODELS*. Springer, 2010, pp. 121–135.
28. D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, "Henshin: A usability-focused framework for EMF model transformation development," in *ICGT*, 2017, pp. 196–208.
29. T. Kehrer, U. Kelter, and G. Taentzer, "A rule-based approach to the semantic lifting of model differences in the context of model versioning," in *ASE*. IEEE Computer Society, 2011, pp. 163–172.
30. D. Strüber, J. Plöger, and V. Acretoaie, "Clone Detection for Graph-Based Model Transformation Languages," in *ICMT*. Springer, 2016, pp. 191–206.
31. D. Strüber, T. Kehrer, T. Arendt, C. Pietsch, and D. Reuling, "Scalability of Model Transformations: Position Paper and Benchmark Set," in *Workshop on Scalable Model Driven Engineering*, 2016, pp. 21–30.
32. T. Thüm, D. Batory, and C. Kästner, "Reasoning about edits to feature models," in *ICSE*, 2009, pp. 254–264.
33. J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr, "Reasoning About Product-Line Evolution Using Complex Feature Model Differences," *Automated Software Engineering*, pp. 1–47, 2015.
34. S. Trujillo, D. Batory, and O. Diaz, "Feature oriented model driven development: A case study for portlets," in *ICSE*. IEEE Computer Society, 2007, pp. 44–53.
35. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," *SPLC*, pp. 77–91, 2010.
36. F. Damiani, R. Hähnle, E. Kamburjan, and M. Lienhardt, "A unified and formal programming model for deltas and traits," in *FASE*. Springer, 2017, pp. 424–441.
37. X. He, Z. Hu, and Y. Liu, "Towards variability management in bidirectional model transformation," in *COMPSAC*, vol. 1. IEEE, 2017, pp. 224–233.
38. E. Biermann, C. Ermel, and G. Taentzer, "Lifting parallel graph transformation concepts to model transformation based on the Eclipse modeling framework," *Electronic Communications of the EASST*, vol. 26, 2010.
39. A. Rensink, "Compositionality in graph transformation," in *ICALP*, 2010, pp. 309–320.
40. A. Ghamarian and A. Rensink, "Generalised compositionality in graph transformation," *ICGT*, pp. 234–248, 2012.
41. G. Perrouin, M. Amrani, M. Acher, B. Combemale, A. Legay, and P.-Y. Schobbens, "Featured model types: towards systematic reuse in modelling language engineering," in *MiSE*. IEEE, 2016, pp. 1–7.

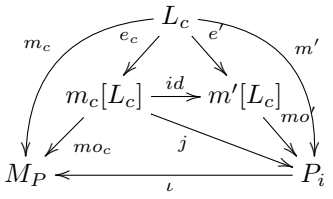
Appendix

Theorem 1 (Equivalence of fully and partially flattened rule applications). *Given a product line P and a variability-based rule $\check{r}$, $Trans_{FF}(P, \check{r}) = Trans_{PF}(P, \check{r})$*

Proof. Without loss of generality, we may assume a fixed product $P_i \in Flat(P)$ with its inclusion $\iota : P_i \to M_P$.

"$\subseteq$" Let the rule application $(P_i \Rightarrow_{r_c, m'} Q_i) \in Trans_{FF}(P, \check{r})$ be given. From the match $m' : L_c \to P_i$ and the inclusion ι, a morphism $m_c : L_c \to M_P$ arises per composition: $m_c = \iota \circ m'$. This morphism induces a variability-based match $\check{m} = (m_c, c)$: Per Thm. 2 in [12], each match of a flat rule r_c induces a variability-based match. We thus can obtain $\check{m}$ in the same way as in [12]. Match m_c needs to fulfill the dangling condition, which is the case: All edges of P_i are contained in M_P. Since m_c maps to the same nodes in M_P as m' does in P_i, all adjacent edges of nodes deleted using m_c are deleted as well.

Finally, m' needs to be equivalent to the morphism obtained from rerouting m_c to P_i. To this end, we consider the epi-mono factorizations of m_c and m', as shown on the right. Since m_c was constructed over m' and inclusion ι, $m_c[L_c]$ and $m'[L_c]$ are the same object, yielding the identity morphism id in the diagram, as well as $e_c = e'$ and $j = mo'$. We have $m' = mo' \circ e' = mo' \circ id \circ e_c = j \circ e_c = reroute(m_c, P_i)$.



"$\supseteq$" Per Def. 6, the rule application $(P_i \Rightarrow_{\check{r}, \check{m}'} Q_i) \in Trans_{PF}(P, \check{r})$ resolves to the rule application $P_i \Rightarrow_{r_c, m_c} Q_i$ for a valid configuration c. Since $r_c \in Flat(\check{r})$ and $P_i \in Flat(P)$, this rule application is contained in $Trans_{FF}(P, \check{r})$.