Transformations of Software Product Lines: A Generalizing Framework based on Category Theory

Gabriele Taentzer[1], Rick Salay[2], Daniel Strüber[3], and Marsha Chechik[2]
[1] Philipps-University Marburg, Germany. Email: taentzer@mathematik.uni-marburg.de
[2] University of Toronto, Canada. Email: {rsalay,chechik}@cs.toronto.edu
[3] University of Koblenz and Landau, Germany. Email: strueber@uni-koblenz.de

***Abstract*—Software product lines are used to manage the development of highly complex software with many variants. In the literature, various forms of rule-based product line modifications have been considered. However, when considered in isolation, their expressiveness for specifying combined modifications of feature models and domain models is limited. In this paper, we present a formal framework for product line transformations that is able to combine several kinds of product line modifications presented in the literature. Moreover, it defines new forms of product line modifications supporting various forms of product lines and transformation rules. Our formalization of product line transformations is based on category theory, and concentrates on properties of product line relations instead of their single elements. Our framework provides improved expressiveness and flexibility of software product line transformations while abstracting from the considered type of model.**

I. Introduction

A *software product line* (SPL) is a portfolio of software products sharing a set of common core assets, while differing in some increments of functionality, often referred to as *features*. SPLs empower enterprises to produce custom-tailored products according to their customers' needs. Due to this key benefit, global players such as Boeing, General Motors, and Toshiba have recently adopted SPL approaches [1].

However, this benefit comes at a price: too much variability in an SPL can give rise to significant complexity. In an SPL with n features, up to 2^n individual products need to be managed and maintained. One of the standard approaches to support developers during these tasks involves the specification of a set of features using *feature models* [2], and the annotation of domain models with presence conditions over these features. Domain models may be provided in typical MDE languages such as UML [3], [4], [5] or petri nets [6], or in source code [7]. Products are produced by *configuring* the features, i.e., switching them on or off, and removing parts of the domain model annotated with inactive features.

To enable the systematic management of an SPL, a variety of transformations needs to be supported. For instance, some transformations aim to only add new products while leaving all of the existing products unchanged. Others aim to affect only the existing products, modifying them in a systematic way. Specifically, the following kinds of transformations have been considered in the literature:

1. *Changes to feature models* aim to support reasoning about additions, deletions, and modifications of features. Thüm et al. [8] distinguish four categories of changes based on their impact to the set of products: refactorings, generalizations, specializations, and arbitrary editing steps. On this basis, Bürdek et al. [9] use model transformation rules to specify high-level editing operations that can be used to support the comprehension of feature model differences. These works are useful to support the evolution of a SPL with versioning tools, yet they do not take the domain model into account.

2. *Lifting of model transformations* [10], [11] aims to make transformation rules from a single-product setting applicable to the entire SPL. The intended effect of lifting is the same as applying the considered rule to each product separately. However, to avoid the combinatorial explosion of enumerating and considering all products, lifting allows the direct application of rules to annotated domain models. Since lifting focuses on domain models, it cannot be used to change the feature model.

3. *SPL refinement* aims to support safe evolution, in the sense that modifications of the SPL have a controlled impact on the existing set of products. Allowed modifications may be restricted so that all products remain unchanged [12], [13], or so that only a subset of the products can be changed [14]. These works provide a form of transformation rules that can change the feature model and restrict the allowed changes on implementation assets, including domain models, but it cannot be used to specify changes of the domain model.

None of the above approaches allows specifying the *combined* transformation of feature models and domain models. However, such combined transformations are actually extremely important in practice, since the addition or deletion of features usually entails the corresponding changes in the domain model. Developers need to know whether such a combined transformation yields a well-formed SPL again. To this end, in the following example, we aim to demonstrate why they would benefit from a means to explicitly specify and reason about combined transformations.

Example. Consider a washing machine controller SPL. Fig. 1 shows a simple feature model, comprising a set of features F and a set of feature constraints Φ, and a state machine representing the domain model. According to the feature model, washers have four features: *Wash*, *Heat*, *Dry*, and *Delay*. In our product portfolio, we want *Wash* to be a mandatory feature of all washers, while a washer can only have either *Heat* and *Delay*; the constraints in Φ are specified accordingly. Conse-

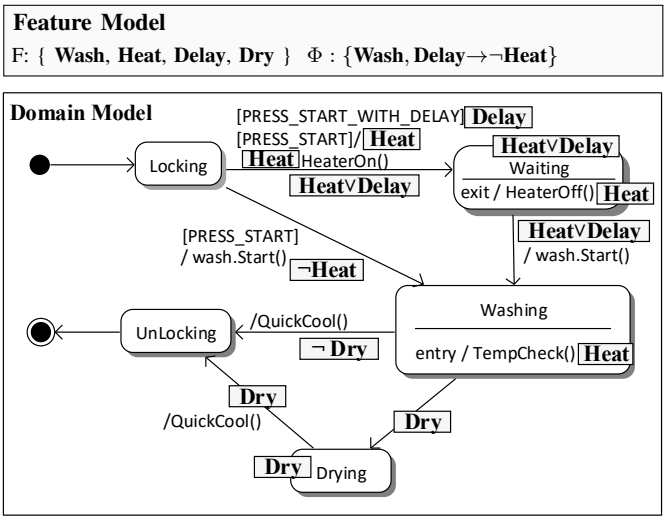

Fig. 1. Washing Machine Controller Product Line: feature model and domain model (adapted from [10]).

quently, the entire product line contains six products. The state machine consists of states with entry and exit actions such as *TempCheck()*, and of transitions. Transitions have an action such as *wash.Start()* and a trigger such as *PRESS_START*, which indicates that a "Start" button was pressed; actions and triggers can be empty. Each of these elements can be annotated with a presence condition to specify the set of products where the element is contained. Consequently, states without presence conditions, such as *Locking*, *Washing* and *UnLocking*, are contained in all products, whereas the states *Drying* and *Waiting* are only present if the feature *Dry* or the feature disjunction *Heat* ∨ *Delay* is active, respectively. In this paper, triggers and actions have the same presence conditions as their containing transitions, unless otherwise specified.

The development and evolution of SPLs can be described systematically using model transformation rules ("rules" in short). A rule consists of a left-hand side (LHS), specifying a pattern to be matched against the input SPL, and a right-hand side (RHS), specifying a desired modification if the LHS matches. It can also optionally contain negative application conditions (NACs) to require non-existence of a certain pattern in the input model. We aim to specify rules that can express patterns and modifications of both the feature model and the domain model. To this end, we introduce rules in which the LHS, RHS, and NACs can be product lines – each has a set F of features, a set Φ of constraints, and an annotated graph, as shown in Fig. 2. Rules can also have parameters, as shown in the brackets within their title bars.

Rule 1 is a simple refactoring rule that collapses two actions of the same name into an entry action of their transitions' common target state, if this state does not already have an entry action. The NAC includes a variable $a1$ to express that the name of the action is irrelevant, that is, $a1$ is a "wildcard". Since this rule does not affect the feature model, F and Φ are empty. Adopting the notion of *lifting* [10], we assume that the effect of applying this rule to the example SPL is the same as applying the domain-model part on each product separately.

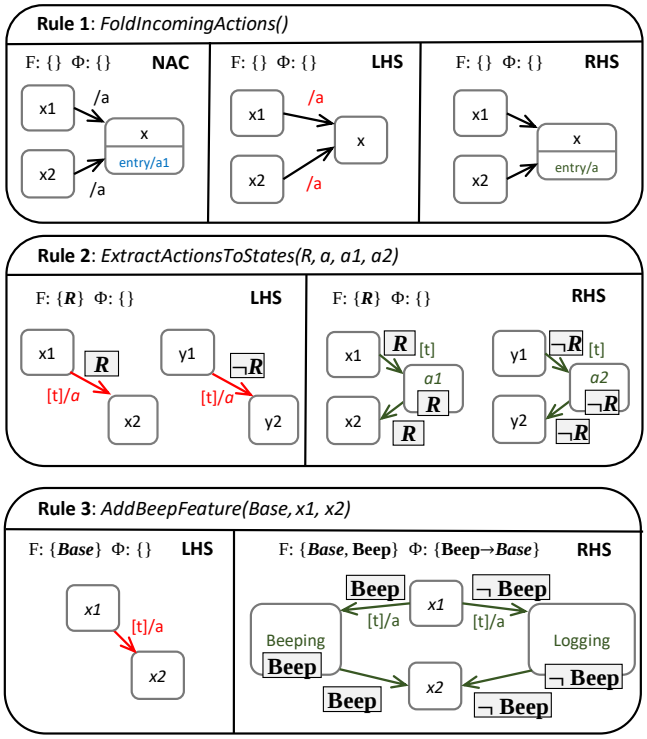

Fig. 2. Transformation rules.

Rule 2 presents an *editing rule*. Such rules have several applications in model-driven engineering: for instance, one can specify the editing operations of a model editor [15], and use them to detect high-level differences between model versions [9]. We would like to specify an editing rule that behaves differently for different products, depending on their features. Feature assignments then represent additional pre- and postconditions, which gives rise to an extended form of lifting to capture this product-wise behavior. In addition, the rule can be used to express safe evolution steps in the context of SPL refinement [14]. The rule's intention is to extract two actions of the same name a into distinct states with different, user-provided names $a1$ and $a2$. It matches two parts of the input domain model that are annotated with a user-specified feature R and its negation. The original transitions are replaced with transitions to the newly created states, which obtain the same, potentially empty, trigger and action, called t and a. For instance, this rule can be applied to the overall product line to turn the *QuickCool()* transitions into states *QuickCooling* (after drying) and *SlowCooling* (after washing).

Rule 3 presents a rule for encapsulating an *evolution step*. By creating a new feature, the rule *specializes* the SPL [8], in the sense that new products are added while also specifying the intended change of the domain model. This rule, in contrast to Rules 1 and 2, refers to a specific feature, *Beep*. To enable a controlled evolution like in SPL refinement [14], the user provides certain restrictions of the rule application as parameters: The feature *Base* is a user-specified parameter that acts as the parent feature of *Beep* – hence the implication in Φ. In the domain model, two states for beeping are inserted between two existing, user-specified states, so that the "beeping" is silent

when *Beep* is not selected. In the example, this rule can be used to introduce loud and silent beeping between washing and unlocking by specifying *Wash* as the base feature, and *Washing* and *Unlocking* as $x1$ and $x2$. The newly created transitions obtain the trigger t and action a of the deleted one. An evolution step such as the one specified by Rule 3 might be reused in SPLs representing other embedded systems, e.g., coffee machines or microwave controllers.

Contributions. To support the specification of and reasoning with such rules, in this paper, we present a formal framework for rule-based transformations of SPLs using category theory. Specifically, we make the following contributions:

1. We formalize the *category of software product lines*. The key benefit of this formalization is that it allows our framework to largely abstract from the type of model being considered, which makes it applicable to a great variety of models, including UML models and Petri nets.
2. We formally define *transformations over software product lines*, and show that our transformations are sound in the sense that for a given matching site of a rule in an input SPL, the rule application yields a well-defined product line as result and this result is unique. We discuss how our framework lays the basis for new kinds of SPL analyses with well-behaved tool support.
3. We demonstrate the *applicability* of our formalization using the example introduced above. We show that our approach supports modifications such as those in the aforementioned works, as well as entirely new kinds of modifications resulting from their combination. Thus, our framework offers users a highly expressive and flexible means for specifying SPL transformations.

The rest of this paper is structured as follows. In Sect. II, for readers not familiar with category theory, we revisit the concepts used in this work in a semi-formal manner. In Sect. III, we define the category of SPLs. In Sect. IV, we introduce our framework for rule-based transformations of SPLs. In Sect. V, we discuss the implications of our framework. In Sect. VI, we conclude, discuss related work, and outline future work.

II. Background

Category theory provides a uniform approach to concepts, constructions, and proofs across a wide range of structures. This property makes category theory very interesting to use it in mathematics and computer science in general, and model-driven engineering (e.g., [16], [17], [18]) in particular. In this section, we give a short and semi-formal introduction into all those notions of category theory we need for our approach. For more details see [19], [20], [21], and [22].

Categories. A *category* **C** is a mathematical structure that has objects collected in Ob_C and morphisms $Mor_C(A, B)$ relating pairs of objects $A, B \in Ob_C$ in some way. There needs to be a composition operation $\circ$ for morphisms $f \in Mor_C(A, B)$ and $g \in Mor_C(B, D)$ as well as an identity morphism id_A for each object $A \in Ob_C$. The composition $\circ$ has to be associative.

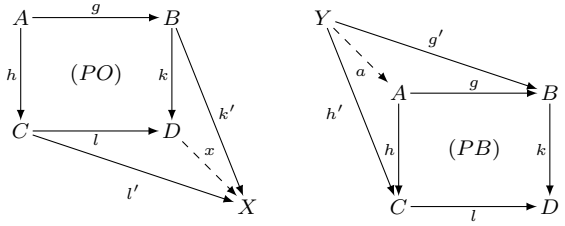

Fig. 3. A schematic depiction of a pushout (left) and a pullback (right).

Examples are the category **Set** of all sets and functions, the category **Rel** of all sets and relations, the category **Poset** of all partially ordered sets and order-preserving mappings, and the category **Graph** of all graphs and graph morphisms being functions between their node and edge sets, resp., such that they are compatible with source and target functions.

There are special types of morphisms: An *isomorphism* is a morphism to which an inverse morphism exists, i.e., composing them in either order leads to identities. Objects related by an isomorphism exhibit exactly the same structure and can thus be considered as equal in many contexts. If we have $m \circ f = m \circ g \implies f = g$ for any two morphisms f and g, m is called *monomorphism*. In the category **Set**, isomorphisms are bijective functions while monomorphisms are injective ones.

Boolean algebras. To represent presence conditions in product lines, we use the *term algebra of propositional formulas*. For its formal definition, we need the signature *BOOL* with the sort *Bool* and the operations *true*, *false*: $\rightarrow$ *Bool*, *not*: *Bool* $\rightarrow$ *Bool*, and *and,or*: *Bool* $\times$ *Bool* $\rightarrow$ *Bool*. All *BOOL*-algebras and homomorphisms, i.e., mappings of algebras along their signature, form a category, called **Alg**(*BOOL*).

Given a set of variables F (later called *features*), there is a special *BOOL*-algebra $T_{BOOL}(F)$ containing all propositional formulas over F. $T_{BOOL}(\emptyset)$ contains terms over *true* and *false* only. $\mathbf{2}$ is the *BOOL*-algebra that just contains *true* and *false*. Given a function f: $F \rightarrow G$ between two variable sets, there is an induced *BOOL*-homomorphism f^*: $T_{BOOL}(F) \rightarrow T_{BOOL}(G)$ between their term algebras. Given a function $\alpha : F \rightarrow \mathbf{2}$, there is a unique *BOOL*-homomorphism α^*: $T_{BOOL}(F) \rightarrow \mathbf{2}$ which lifts this assignment to all terms over F. And finally, there is a unique assignment $\alpha_\emptyset : T_{BOOL}(\emptyset) \rightarrow \mathbf{2}$ such that $\alpha^* \circ f^*_\emptyset = \alpha_\emptyset$ for the unique map $f_\emptyset : \emptyset \rightarrow F$ and all α.

Initial objects, pushouts and pullbacks. An object I of category **C** is called *initial* if for each object A of **C**, there exists a unique morphism $i_A : I \rightarrow A$. In the category **Set**, the initial object is the empty set since it can be uniquely embedded into any other set.

A *pushout* can be considered as a kind of union of two objects over a common one. Given two morphisms g: $A \rightarrow B$ and h: $A \rightarrow C$, a pushout, if it exists, consists of an object D and two morphisms $k : B \rightarrow D$ and $l : C \rightarrow D$ such that $k \circ g = l \circ h$ and the following property holds: If there are morphisms $k' : B \rightarrow X$ and $l' : C \rightarrow X$ with $k' \circ g = l' \circ h$, then there is a unique morphism $x : D \rightarrow X$ with $x \circ k = k'$

and $x \circ l = l'$ (see the left diagram in Fig. 3). These properties ensure that the union of the considered objects is large enough but not too large. A *colimit* can be considered as an even more general kind of union of a set of inter-related objects.

Reversing the direction of all morphisms, a *pullback* can be seen as a generalized intersection of two objects over a common object. Given two morphisms $k : B \to D$ and $l : C \to D$, a pullback consists of an object A and morphisms g: $A \to B$ and h: $A \to C$ if $k \circ g = l \circ h$ and the following property holds: If there are morphisms $g' : Y \to B$ and $h' : Y \to C$ with $k \circ g' = l \circ h'$, then there is a unique morphism $a : Y \to A$ with $g \circ a = g'$ and $h \circ a = h'$ (see the right diagram in Fig. 3). These properties ensure that the intersection of the considered objects covers their common part completely but not more.

In the category **Set**, if a morphism g is injective, the pushout object is $D = C \cup (B - g(A))$. Since a pushout is unique up to isomorphism, any set isomorphic to D would also be a pushout object. A pullback object, for l being injective, is constructed by $A = k(B) \cap l(C)$. In the category **Graph**, pushouts and pullbacks can be constructed component-wise on node and edge sets.

Given a morphism $l : C \to D$, an *initial pushout* is the smallest possible pushout such that l is one of the pushout morphisms. In **Set**, an initial pushout is constructed over the morphisms $\emptyset \to C$ and $\emptyset \to (D - l(C))$. In **Graph**, A might contain some boundary nodes needed to glue C with $D - l(C)$ along $A \to C$ and $A \to (D - l(C))$.

A *pushout complement* can be considered as a kind of generalized difference between two objects. It is shown in [22] that pushout complements are unique, i.e., given monomorphisms $A \to C$ and morphism $C \to D$, there is at most one complement, up to an isomorphism, B with $A \to B$ and $B \to D$ such that the left diagram in Fig. 4 is a pushout.

$\mathcal{M}$-adhesive categories. Product lines form a lattice of objects; to be able to reason about unions of objects and subobjects in a general setting, we recall $\mathcal{M}$-adhesive categories from [21] and [22]. A category **C** with a morphism class $\mathcal{M}$ is an *$\mathcal{M}$-adhesive category* if the following properties hold:

1. $\mathcal{M}$ is a class of monomorphisms closed under isomorphisms (f isomorphism implies that $f \in \mathcal{M}$), composition ($f, g \in \mathcal{M}$ implies $g \circ f \in \mathcal{M}$), and decomposition ($g \circ f, g \in \mathcal{M}$ implies $f \in \mathcal{M}$).
2. **C** has pushouts and pullbacks along $\mathcal{M}$-morphisms, i.e., pushouts and pullbacks, where at least one of the given morphisms is in $\mathcal{M}$, and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks, i.e., given a pushout like the left diagram in Fig. 4, $m \in \mathcal{M}$ implies $n \in \mathcal{M}$ and, given a pullback (1), $n \in \mathcal{M}$ implies $m \in \mathcal{M}$.
3. Pushouts in **C** along $\mathcal{M}$-morphisms are stable under pullbacks, i.e., for any commutative cube in **C** where we have the pushout with $m \in \mathcal{M}$ in the bottom, $b, c, d \in \mathcal{M}$, and pullbacks on all sides, the top is a pushout as well. Such a cube (Fig. 4) is called *$\mathcal{M}$-Van Kampen (VK) square*.

Examples for $\mathcal{M}$-adhesive categories are sets with injective functions, graphs with injective graph morphisms, several variants of graphs like typed and typed attributed graphs with special forms of injective graph morphisms, place-transition (Petri) nets with injective net morphisms, and algebraic specifications with restricted injective specification morphisms. In these categories, a $\mathcal{M}$-VK square characterizes the property that a union of objects can be reduced to a union of subobjects.

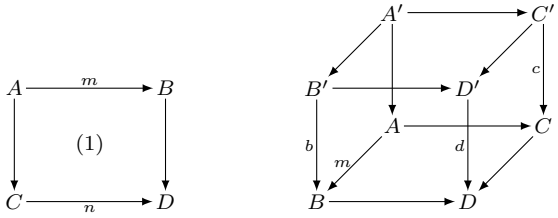

Fig. 4. A schematic depiction of an $\mathcal{M}$-VK square.

III. Category of product lines

In this section, we formally define product lines and their relations. To be able to largely abstract from the kind of domain models considered in product lines, we characterize the constraints that have to be fulfilled by structures acting as domain models. Product line relations such as embeddings or refinements can be expressed by product line morphisms. The soundness of these definitions is established by our first main result showing that product lines and product line morphisms form a category. Our transformation approach for product lines is based on gluing (or union) of product lines without replicating their common parts. This glueing can be formally characterized as a pushout in this new category.

In the following, we define the underlying category **Model** of finite domain models, generalized product lines, relations between product lines called product line morphisms, and finally, the category of product lines over models. We use basic notions from category theory summarized in Sect. II.

A. Domain models

The underlying domain models of product lines are not restricted to a specific kind of structures; we just require that they come from a category **Model** fulfilling the following constraints:

Definition 1 (Category **Model***):* Category **Model** can be any finitary $\mathcal{M}$-adhesive category that has initial pushouts. An object of **Model** is referred to as *model*.

Examples of **Model** are the categories of finite sets, Petri nets, labeled graphs, typed graphs, and typed, attributed graphs, as shown in [24], [23].

We define some notions and terminology generalized from set theory to be used in the remainder of the paper.

Definition 2 (Notation and terminology): Let models M, N, and M_s be given.

1. We can put a preorder on $\mathcal{M}$-morphisms into a model M by defining an $\mathcal{M}$-morphism $a : A \to M$ to be less than or equal to an $\mathcal{M}$-morphism $b : B \to M$ iff there is a

morphism $c : A \to B$ with $b \circ c = a$. A *submodel* of M is an equivalence class w.r.t. the equivalence generated by this preorder. Submodel $a : A \to M$ is also denoted by $A \subseteq M$.

2. A submodel M_C of M is a *complement* of a submodel $M_S \subseteq M$ if there is an initial pushout over an $\mathcal{M}$-morphism $s : M_S \to M$ of the corresponding equivalence class. As part of the result, there is an $\mathcal{M}$-morphism $M_C \to M$ as second pushout morphism. M_C is also denoted as $M - s(M_S)$.
3. $\mathcal{P}(M)$ (called *a power set* of M) consists of all submodels of M. It forms a complete (finite) lattice w.r.t. submodel union and intersection.
4. Given an $\mathcal{M}$-morphism $i : M_S \to M$ and a morphism $m : M \to N$, the composed morphism $m \circ i$ is called *restriction* and is denoted by $m_{|M_S}$.
5. Given two submodels S_1, S_2 of M, the union of S_1 and S_2, written $S_1 \cup S_2$, is constructed by the pullback of both submodels and taking the pushout of this pullback. More generally, given a finite set S of submodels of M, the union of submodels in S, denoted by $\bigcup S$, is constructed by taking the pullbacks of submodels pairwise and taking the colimit of all these pullbacks, i.e., unifying all submodels along their pairwise overlap.

In our running example, we represent Statecharts as graph-based models. As pointed out above, typed attributed graphs are shown to be $\mathcal{M}$-adhesive [24], [22]. Submodels are graphs that can be embedded into larger graphs by injective graph morphisms. The *complement* C of a graph S being a submodel of G contains all graph elements of G not in the image of S. Since the set of these elements usually does not yield a proper graph, so-called *boundary nodes* have to be added, i.e., there is a graph B consisting of boundary nodes only that can be embedded into both S and C.

B. Product lines and product line morphisms

In annotative product lines, each *element* of the domain model is annotated with a Boolean presence condition. We have to annotate the domain model in the right way, e.g., annotating a state but not its adjacent transitions would lead to products with dangling transitions. For a structure-compatible annotation, we say that each *submodel* of the domain model is annotated with a Boolean presence condition. The submodel approach forms the basis for our definition as the power set $\mathcal{P}(M_P)$ of a domain model M_P forms a *BOOL*-algebra which can be mapped to presence conditions by a homomorphism f_P such that, whenever $M_1 \subseteq M_2 \subseteq M_P$, then $f_P(M_2) \implies f_P(M_1)$.

Definition 3 (Product line): Given a category **Model** as in Def. 1, a product line $P = (F_P, \Phi_P, M_P, f_P)$ over **Model** consists of the following parts:

1. a *feature model* that consists of a set F_P of features, and a set of *feature constraints* $\Phi_P \subseteq T_{BOOL}(F_P)$,
2. a *domain model* M_P being an object of category **Model**,
3. a *set of presence conditions* expressed as a *BOOL*-homomorphism f_P: $\mathcal{P}(M_P) \to T_{\mathit{BOOL}}(F_P)$.

P is called *featureless* if $F_P = \emptyset$, $\Phi_P = \{\}$, and $f_P(M) =$ *true* for all $M \subseteq M_P$, and *element-free* if M_P is empty.

Note that a featureless product line can be defined straightforwardly from any domain model, and an element-free product line from any feature model. Note further that $f_P(M_P) = \Phi_P$ does not have to hold.

Example 1 (Product line): Fig. 1 shows an example of a product line. The feature set, the set Φ of feature constraints, and the domain model, a state machine, are denoted directly. Various elements of the state machine in Fig. 1 are annotated with presence conditions. Elements with the same presence conditions, potentially completed to a graph by boundary nodes, form submodels being mapped to the corresponding Boolean terms over the given feature set. For example, the state *Drying* and the transitions between states *Washing* and *Drying* as well as *Drying* and *Unlocking* are annotated with the condition *Dry*, i.e., the submodel consisting of these three states and two transitions is mapped to a term *Dry*.

Given a product line, the set of all of its products can be derived by considering all possible feature assignments.

Definition 4 (Feature configuration): A *valid feature configuration* of a product line P is an assignment α: $F_P \to \mathbf{2}$ such that $\alpha^*(\phi)$ = *true*, $\forall \phi \in \Phi_P$. The set of all valid feature configurations in P is denoted by $\mathit{Conf}(P)$.

Definition 5 (Product): Given a product line $P = (F_P, \Phi_P, M_P, f_P)$,

1. a product M is derived from P under the valid feature configuration α if M is the union of all those models $M' \subseteq M_P$ for which $f_P(M')$ is satisfiable by α^*, i.e., $M = \bigcup\{M' \subseteq M_P | \alpha^*(f_P(M')) = \textit{true}$ and α is valid$\}$, and
2. the set *Prod*(P) of all products of P with their configurations is defined as *Prod*(P) = $\{(M, \alpha) | M$ is a product of P under α and α is valid$\}$.

Note that we do not require *Prod*(P) to have at least one product.

Example 2 (Product): Given the product line P in Example 1 and a feature assignment α(*Wash*) = *true* and α(*Heat*) = α(*Delay*) = α(*Dry*) = *false*, the corresponding product (M, α) consists of the start and end states, states *Locking*, *Washing*, and *Unlocking*, and transitions between them.

Our formalization is essentially based on relations between product lines which are expressed by morphisms. In the following, we define a morphism m from product line P to product line Q as a pair of mappings between their feature sets and their domain models. Such a morphism has to ensure that each product of P is mapped to a submodel of a product of Q assuming compatible feature assignments.

Definition 6 (Product line morphism): Given two product lines $P = (F_P, \Phi_P, M_P, f_P)$ and $Q = (F_Q, \Phi_Q, M_Q, f_Q)$

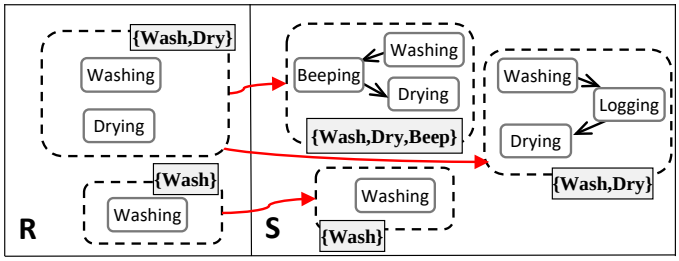

Fig. 5. Product line morphism (product-wise representation).

over category **Model**, a *product line morphism* m: $P \to Q$ over category **Model** is defined by $m = (m_F, m_M)$ with

1. m_F: $F_P \to F_Q$ being a function over feature sets F_P and F_Q such that $\Phi_Q \implies m_F^*(\Phi_P)$ and
2. m_M: $M_P \to M_Q$ being a **Model**-morphism such that $\forall (M^P, \alpha_P) \in \mathit{Prod}(P) : \exists (M^Q, \alpha_Q) \in \mathit{Prod}(Q)$ with $m_M(M^P) \subseteq M^Q$ and $\alpha_Q \circ m_F = \alpha_P$.

A product line morphism $m = (m_F, m_M)$ is *feature-preserving* (*feature-refining*) if m_F is bijective (injective). It is *constraint-strict* if it is feature-refining and $\Phi_Q \Leftrightarrow m_F^*(\Phi_P)$.

Example 3 (Product line morphism): Fig. 5 shows two example product lines R and S based on the running example. For illustration purposes, the product lines are shown in a product-wise representation; an annotated representation of R can be derived based on Fig. 1. The relation of R and S can be specified by a product line morphism $n = (n_F, n_M)$. Since the features of R are a subset of those of S, n_F is injective. The **Model**-morphism n_M embeds R's domain model, comprising the union of all shown states and transitions, into that of S. As shown in the figure, each product in R is mapped to a product in S potentially extending its configuration and its submodel, thus satisfying the condition.

Definition 7 (Special product line morphisms): Given a product line morphism g: $P \to Q$ with $g = (g_F, g_M)$, it is *injective* (*bijective*) if g_F is injective (bijective) and g_M is an $\mathcal{M}$-morphism (isomorphism).

Product line morphisms can be used to embed one product line into another, to add or rename features or domain model elements, or to glue them together:

1. The product line morphism $m : P \to Q$ between featureless product lines P and Q is clearly determined by the **Model**-morphism m_M. m_F is the empty morphism with $\mathit{true} \implies \mathit{true}$.
2. A bijective m_F allows a renaming of features.
3. Since m_F may be non-injective, it allows to glue features.
4. Since m_F may be non-surjective, it allows to introduce new features.

We show later that product line transformations can also be used to remove features and domain model elements. Next, we show that product lines and product line morphisms as defined above form a category. To do so, we first define identity morphisms on product lines as well as the composition operator for product line morphisms.

Proposition 1 (Category ProdLine): Product lines and product line morphisms over a given category **Model** with the identities and composition operator as defined above form a category, called $\mathbf{ProdLine_{Model}}$.

Proof sketch: There is an identity morphism for each product line P being defined by the identities on the feature set and the domain model of P. The composition of two product line morphisms g: $P \to Q$ and h: $Q \to R$ defined component-wise on functions and **Model**-morphisms again yields a product line morphism. Associativity of product line morphisms holds due to the associativity of the composition of functions and morphisms. The full proof is given in [26].

C. Pushouts in the category of product lines

Product line transformations as we consider them below are largely based on the gluing of product lines, i.e., their union. As pointed out above, pushouts can characterize a kind of union of two structures over a common one. Hence, we need to clarify how pushouts of product lines look like. In the following, we only consider the case where one product line morphism is injective. This setting is already sufficient to base product line transformations on pushouts later on.

Proposition 2 (Pushout in category $\mathbf{ProdLine_{Model}}$): Given two product line morphisms g: $P \to Q$, h: $P \to R$ over **Model** with $g = (g_F, g_M)$ being injective and $h = (h_F, h_M)$, there is a pushout $(S, k : Q \to S, l : R \to S)$, where $S = (F_S, \Phi_S, M_S, f_S)$ is constructed as follows:

1. *Pushout of feature sets:* (F_S, k_F, l_F) is the pushout of (g_F, h_F) in **Set**. Let $F_S = F_R \cup (F_Q - g_F(F_P))$.
2. *Construction of feature constraint set:* $\Phi_S = l_F^*(\Phi_R) \cup k_F^*(\Phi_Q)$.
3. *Pushout of domain model:* (M_S, k_M, l_M) is the pushout of (g_M, h_M) in **Model**. Let $M_S = M_R \cup (M_Q - g_M(M_P))$ (with g_M being an $\mathcal{M}$-morphism). (Compare with Def. 2, items 2 and 5.)
4. *Construction of presence conditions:* To construct f_S, we distribute each submodel M^S of M_S along its pushout construction yielding $M^S = M^R \cup (M^Q - g_M(M^P))$.
   (a) *Product comes from R:* $f_S(M^S) = l_F^*(f_R(M^R))$ if (M^R, α_R) is a product while (M^P, α_P) is not. All (M^S, α_S) are products for which a valid feature configuration is given by $\forall f \in F_S$: $\alpha_S(f) = \alpha_R(f)$ for $f \in F_R$, and $\alpha_S(f) \in \mathit{Conf}(Q)$ otherwise.
   (b) *Product comes from Q:* $f_S(M^S) = k_F^*(f_Q(M^Q))$ if (M^Q, α_Q) is a product while (M^P, α_P) is not. All (M^S, α_S) are products with valid feature configuration $\forall f \in F_S$: $\alpha_S(f) = \alpha_Q(f)$ for $f \in F_Q$, and $\alpha_S(f) \in \mathit{Conf}(R)$ otherwise.
   (c) *Products from R and Q are unified:* $f_S(M^S) = l_F^*(f_R(M^R)) \wedge k_F^*(f_Q(M^Q))$ if (M^R, α_R), (M^Q, α_Q) and (M^P, α_P) are products. All (M^S, α_S) are products with valid feature configuration $\forall f \in F_S$: $\alpha_S(f) = \alpha_R(f)$ for $f \in F_R$, and $\alpha_S(f) = \alpha_Q(f)$ otherwise.
   (d) For all remaining submodels $M^S \in \mathcal{P}(M_S)$:

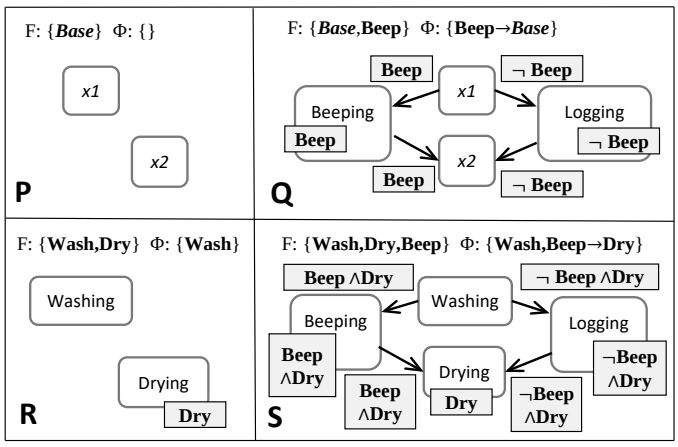

Fig. 6. Pushout for applying a part of Rule 3 to a part of the washer SPL.

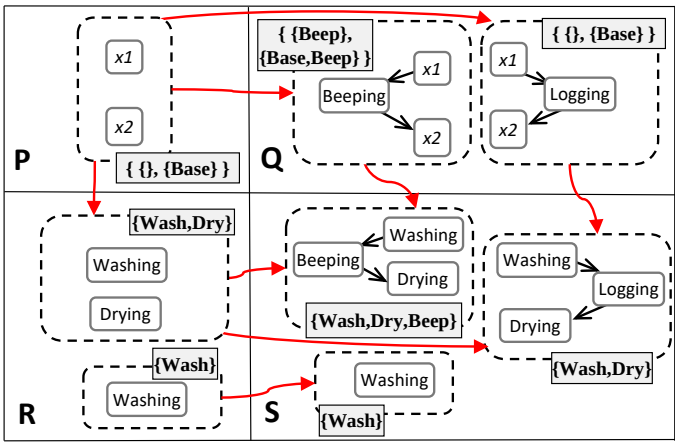

Fig. 7. Alternative, *product-wise* representation of the pushout.

$f_S(M^S) = \bigvee\{f_S(M) | M \text{ is product } \wedge\ M^S \subseteq M\}$. Note that $\bigvee \emptyset = false$.

Proof sketch: The pushout is computed component-wise. Step 1 computes the pushout of feature sets. Step 2 combines sets of feature constraints. Step 3 computes the pushout of the domain models. Finally, in Step 4, the presence conditions are computed for each submodel. Since category **Model** is $\mathcal{M}$-adhesive, M^S can always be split up as shown above. The products of the pushout product line are constructed from the products of the original product lines: either two products stemming from the two original product lines are unified over a common subproduct, or a product of an original product line is "adopted" (but it can be renamed and extended).

We have to show that all sub-models of the pushout object model are equipped with a unique presence condition. Moreover, all newly constructed mappings have to be product line morphisms. Furthermore, the pushout property has to be shown. The complete proof is given in [26].

Example 4 (Pushout construction): Fig. 6 shows four product lines P, Q, R, and S participating in a pushout. While P has just one product, Q and R have two products each, and S has three products. Note that R just shows the part of the SPL in Fig. 1 which is needed to construct a pushout, and, for conciseness, we omit the treatment of the transitions' effects. We later show an extended version of this example.

The product line morphism $P \to Q$ summarizes all of the creating actions of Rule 3 in Fig. 2: a new concrete feature *Beep* is added together with a new implication; in the domain model, two states with their adjacent transitions are added. The product line morphism $P \to R$ maps the preserved part of Rule 3 to a submodel of the state machine in Fig. 1. The feature variable *Base* is mapped to the feature *Wash*. In the domain model, state $x1$ is mapped to state *Washing* while state $x2$ is mapped to *Drying*. The pushout glues Q with R over their common subproduct line P. According to steps 1 and 2 of the construction in Prop. 2, the feature sets and their constraint sets are unified over the common feature *Base*. In step 3, the domain models are glued over states $x1$ and $x2$. Domain model elements are annotated along products in step 4.

To understand this annotation better, we present the (parts of) products participating in this pushout construction in Fig. 7. Since the product where just the feature *Wash* is selected consists of the state *Washing* only, there is no source product that can be mapped to it. Hence, this product is just taken over into the pushout product line S according to step 4(a). Since both products in Q contain the state $x2$ and since the other product in R is obtained if the feature *Dry* is selected as well, the two remaining products in S require that *Dry* is selected. Hence, the corresponding products of R and Q are glued over the only product in P per step 4(c). Note that some of the considered products are obtained from multiple configurations, e.g., the one in P is obtained from $\{\}$ and $\{Base\}$.

Since at least one input morphism of the pushout construction is general, multiple distinct features or domain model elements of one SPL may be mapped to the same feature or elements in the pushout SPL.

Summary. In this section, we have accomplished our first goal: to set up a framework for product lines over a variety of possible domain models, not only UML-like but also other kinds as well such as Petri nets or graph structures deduced from program code. Def. 1 specifies the constraints that have to be fulfilled for such domain models. On this basis, we defined product lines and their relations expressed as product line morphisms. Prop. 1 shows that our definitions of product lines and product line morphisms are sound since they form a category. As a foundation for product line transformation, we presented a construction for the union of two product lines over a common one. This construction is sound since it can be characterized as a pushout, as shown in Prop. 2.

IV. Transformation of Product Lines

In this section, we present the new, general form of rule-based product line transformations. They allow to modify feature models and domain models in a single transformation step; they capture all of the examples presented in Sect. I. This form of product line transformation is a so-called *gluing approach* since new elements are glued to the existing ones. Due to its formalization based on category theory, we can show

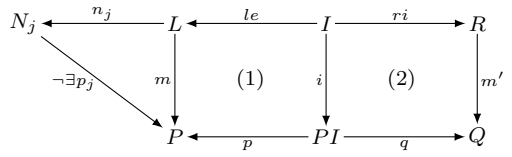

Fig. 8. A schematic depiction of rule application.

that, given a rule and a match, the transformation (a) always has a result, and (b) this result is unique (up to isomorphism).

Definition 8 (Transformation rule): A *(product line transformation) rule* r is defined by $r = (L \xleftarrow{le} I \xrightarrow{ri} R, NAC)$, where L, I, and R are product lines over the category **Model**, le and ri are injective product line morphisms, and NAC is a (potentially empty) set of negative application conditions defined by injective product line morphisms $L \xrightarrow{n_j} N_j$ with $j \in J$.

Transformation rules defined above remove features if le_F is not an isomorphism and add them if re_F not an isomorphism. Feature constraints may be adapted as well. Domain model elements are removed from (added to) a product line if le_M (re_M) is not an isomorphism. Note that gluing and splitting of feature models or domain models is not supported by our definition of rules yet.

Example 5 (Transformation rules): Rule 3 in Fig. 2 directly uses product lines in its LHS and RHS. Product line I is given only implicitly as the intersection of LHS and RHS consisting of the feature set $\{Base\}$, an empty constraint set, and states $x1$ and $x2$ in the domain model. Its embeddings into LHS and RHS are straightforward, and NAC is empty.

Definition 9 (Rule match): Given a rule r defined as above and a product line P over the category **Model** (as defined in Def. 1), a *match* is a product line morphism $m : L \to P$ fulfilling the following conditions:

1. the PO-complement of le_M and m_M in the category **Model** exists, and
2. $\forall N_j \in NAC$: There does not exist an injective product line morphism $p_j : N_j \to P$ with $p_j \circ n_j = m$.

Since the rule match may be non-injective, two separate features or domain model elements in the LHS may be mapped to the same feature or domain model element in the product line P. NACs can be used to disallow the existence of certain features and domain model patterns.

Definition 10 (Rule application): Given a rule r and a match $m : L \to P$ as defined above, a *rule application* $P \Longrightarrow_{r,m} Q$, also called a *transformation (step)*, is given by the diagram in Fig. 8 where (1) and (2) are pushouts in $\mathbf{ProdLine}_{\mathbf{Model}}$.

Rule r is applicable at match m if (1) in Fig. 8 is a pushout in the category $\mathbf{ProdLine}_{\mathbf{Model}}$. Given r and m, this pushout can be constructed by the pushout (PO) complement construction below. A PO-complement over product lines can be considered as the difference between product lines. For all products of P to which a product of L is mapped, the difference product is constructed. All other products of P are just copied to PI.

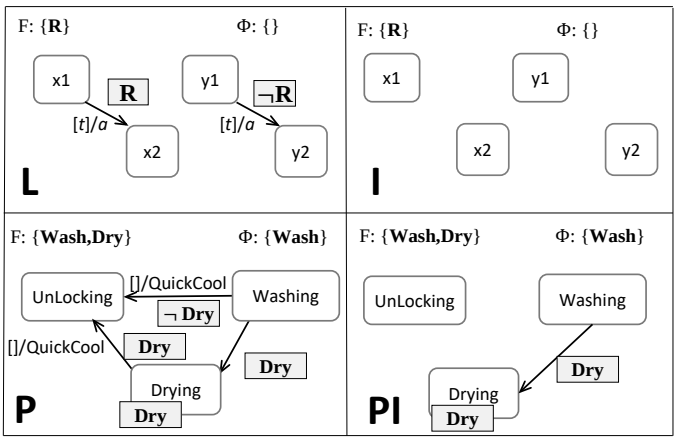

Fig. 9. Pushout complement for applying Rule 2 to a part of the washer SPL.

Proposition 3 (Existence and uniqueness of PO-complement): For a rule r and a match $m : L \to P$ as defined above, the product line $PI = (F_{PI}, \Phi_{PI}, M_{PI}, f_{PI})$ with a pushout (1) exists and is unique iff M_{PI} is the *PO-complement* of le_M and m_M in the category **Model**. PI is constructed as follows:

1. $F_{PI} = F_P - m_F(F_L - F_I)$,
2. $\Phi_{PI} = \Phi_P - m_F^*(\Phi_L - \Phi_I)$,
3. M_{PI} is the unique PO-complement of le_M and m_M. Without loss of generality, $M_{PI} = M_P - m_M(M_L - M_I)$.
4. $\forall M \in \mathcal{P}(M_{PI})$: $f_{PI}(M) = f_P(p_M(M))$

Proof sketch: We have to show that the construction of PI as presented above is always possible and yields a product line. Morphism p is constructed component-wise yielding an injective product line morphism while morphism $i = (i_F, i_M)$ is constructed by $i_F = m_{F_{|F_I}}$ and i_M being defined by the PO-complement construction in **Model**. Moreover, we have to show that there exists no other PO-complement for the given rule and match. The complete proof is given in [26].

Example 6 (PO complement construction): Fig. 9 shows an example of a pushout complement as Rule 2 (depicted in Fig. 2) is applied to a part of the washing machine PL in Fig. 1. The rule match $L \to P$ maps the feature R to *Dry*, $x1$ to *Drying*, $y1$ to *Washing*, and $x2, y2$ to *Unlocking*. The transitions and their actions are mapped accordingly. Note that this mapping is non-injective. The category of typed, attributed graphs and graph morphisms has PO-complements whenever the match does not delete nodes so that context edges may dangle. Since transitions, i.e., edges, are the only elements we can delete, this condition is fulfilled, and thus the PO-complement of the underlying domain models exists.

Since the rule does not change the feature model, this model is not changed in the rule application as well. The domain model of PI still has the same states as P since none of the states are deleted by the rule. Two transitions, however, are deleted. The remaining domain model elements are annotated as in P.

Due to our formalization of product line transformations, we can show that transformation steps are sound, i.e., given a

rule with a match, there exists a well-defined transformation result and this result is unique.

Corollary 1 (Existence and uniqueness of transformations): Given a rule $r = (L \xleftarrow{le} I \xrightarrow{ri} R, NAC)$ and a match $m : L \to P$ in the category $\mathbf{ProdLine}_{\mathbf{Model}}$, the rule application $P \Longrightarrow_{r,m} Q$ exists and is unique up to an isomorphism.

Proof: Because of Prop. 3, there exists a unique PO-complement (PI, p, i) as constructed above. Moreover, the pushout (Q, q, m') is unique up to an isomorphism, due to pushout properties.

Illustration. We reiterate the example introduced in Sect. I in more detail. Fig. 10 shows the application of the original rules to suitable excerpts of the washing machine product line. (1) For rule *FoldIncomingActions*, the selected excerpt includes three products, arising from the selection of feature *Delay*, *Heat*, or neither of them. Since the pair of *wash.start()* actions is only present in the *Delay* product, this is the only product where L matches P. Subsequently, during the transformation step, the left of the two actions is removed and the right one is assigned a new presence condition *Heat*, using the pushout complement of P and I over L. In turn, based on the pushout of PI and R over I, the *Washing* state obtains an equivalent entry action, which, however, is only present in the *Delay* product. (2) The deletion part of *ExtractActionToStates*'s application to the example was already presented to illustrate Prop. 3. Here we also see the creation part. The newly created states $a1$ and $a2$ are assigned the names *SlowCooling* and *DryCooling* as specified by the user, and each of them is being connected to the existing states via transitions. (3) The creation part of *AddBeepFeature*'s application was discussed above, but here, we also see the prequel to the creation: The user has specified *Washing* and *Drying* as values for $x1$ and $x2$, respectively. Consequently, applying the pushout complement to P and I over L removes the transition between these states.

Summary. In this section, we have accomplished our second goal – to define sound transformations of product lines which allow to combine transformations of feature models and of domain models. Since a transformation step is characterized as a double-pushout (see Def. 10), we show in Cor. 1 that, given a rule with a match, the transformation results in a well-defined unique product line. This result relies on characterizing the difference construction as a pushout complement (see Prop. 3).

V. Discussion

Generalizing existing approaches. We now discuss how our framework supports the kinds of transformations discussed in Sect. I. Lifting [10] and feature-model editing [8] are special cases of our SPL transformations, since the "vanilla" transformation rules considered in lifting can be represented as featureless rules, and feature-model editing rules can be represented as element-free rules. To enable safe evolution, SPL refinement [12], [14] provides a catalog of rules (called *templates* in these works) to change the feature model while restricting the allowed changes to the included domain models and asset mappings. For most of these rules – *replace feature expression*, *add mandatory feature*, *remove feature*, *change asset*, *add asset*, *add optional feature* – one can use our framework to specify the intended change of the domain model as well. An exception is the *splitting* of assets [12]. Splitting is not directly possible in our framework as it would require non-injective morphisms in the rules, whereby we only support non-injective matching. As a workaround, splitting can be mimicked by creating a separate feature and reassigning the affected parts of the domain model to it.

The notion of a feature model in our work is somewhat simplified: we represent feature hierarchies in terms of constraints; however, [27] presents an efficient algorithm for deriving full-fledged feature models from these representations.

Towards new SPL analyses and usage scenarios. Our work lays the basis for a variety of new SPL analyses, including conflict and dependency analysis, confluence analysis and termination checks for SPL transformations. Possible usage scenarios for these analyses are (1) bringing transformations into the right order (for handling of transformation chains); (2) composing several transformations to more complex ones (for recurring transformations) and (3) coordination of processes (for SPL development in larger teams). We aim to implement the new analyses on top of the Henshin model transformation language [28], [29]. These analyses also depend on having additional formal results such as the Local-Church-Rosser, Parallelism, Concurrency, and Amalgamation Theorems for parallel and sequential executions of transformations, which are available for $\mathcal{M}$-adhesive categories [21], [22]. We leave showing that the category of product lines is an $\mathcal{M}$-adhesive category for future work.

VI. Related Work and Conclusion

Having discussed the most closely related work in Sect. V, we now discuss further related work.

First, management of a product line involves certain transformations, such as the lifting of single-product rules to an entire SPL [10], [30], and the configuration of an SPL representation to produce individual products [3], [31], [32]. *Compositional approaches* manage an SPL by splitting it into a core product and a set of changes. Batory et al. [16] have investigated a categorical underpinning of such approaches. However, their representation of transformations as *arrows* can only add new elements to the core product. The *delta modeling* approach [33], [34] is more flexible since the changes encapsulated in a delta may also include deletions and modifications. *Abstract delta modeling* [35] introduces a notion of conflict-resolving deltas to deal with situations with multiple conflicting deltas. Evolution histories of SPLs can be expressed as a set of higher-order deltas [36]. Deltas are similar to rules – they encapsulate changes to domain models. However, the goal of delta modeling is to facilitate derivation of products for a fixed feature model, and our approach supports systematic modifications of the feature model.

Second, some related work is concerned with SPL refactoring for achieving a higher-level goal, e.g., to extract a managed

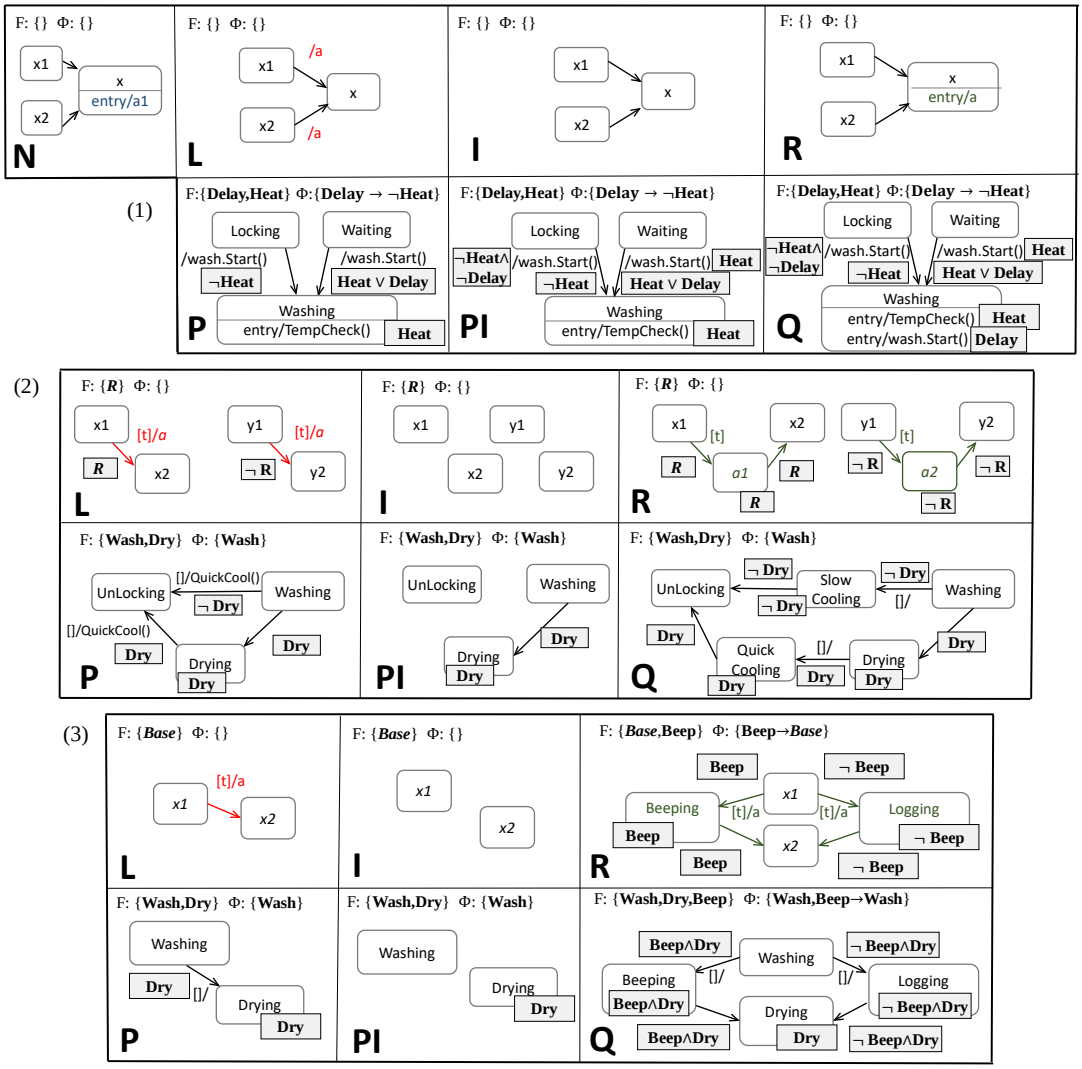

Fig. 10. Transformation of product lines by example: Rules *FoldIncomingActions* (1), *ExtractActionsToStates* (2), and *AddBeepFeature* (3) are applied to excerpts of the washing machine product line.

product line from a set of separate products [37], or to extend the standard object-oriented refactorings to feature-oriented SPLs [38]. Such refactorings can be composed of atomic rule-based editing steps, such as those introduced in this paper.

Third, some works are concerned with the internal variability of transformation rules. *Variability-based model transformation* [39], [40], [41], [42] aims to improve the conciseness and performance of syntactically similar rules by expressing variability explicitly. Instead of SPL transformations, this work focuses on the management of "transformation product lines".

To summarize, in this paper, we presented a formalization of SPLs and their rule-based transformations. Specifically, we defined the generalized form of an annotative SPL using Category Theory, and characterized the type of artifacts over which it can be defined. Then, we formally defined transformation rules of such SPLs using the double pushout approach and gave existence and uniqueness results to show its soundness. The formalization achieves three objectives: first, it provides a systematic and consistent way of defining product lines over a broad class of software artifacts; second, it extends existing work on SPL transformations by allowing richer combinations of changes that transformation rules can make; and third, it provides the foundation for defining general tools and techniques for the development and analysis of SPL transformations.

Each of these objectives points to future work. We have illustrated our formalization on attributed graphs using example rules over an SPL of state machines; however, we plan to demonstrate the generality of the approach by applying it to diverse types of software artifacts. Our approach was motivated, in part, by three threads of research in the SPL transformation literature but we fell short of fully *unifying* them: our approach currently does not handle rules that glue or split product lines. This extension is left to future work. Finally, we plan to use the theoretical foundation of $\mathcal{M}$-adhesive categories on which we based our work, to build up a rich theory of SPL transformations and integrate it into tooling to support sound SPL transformation development.

Acknowledgements. We wish to thank Jens Kosiol and the anonymous reviewers for their constructive comments.

References

[1] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[2] 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., Software Engineering Inst., Tech. Rep., 1990.

[3] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *Proc. of ICGPCE'05*. Springer, 2005, pp. 422–437.

[4] Ø. Haugen, A. Wasowski, and K. Czarnecki, "CVL: Common Variability Language," in *Proc. of SPLC'12*, 2012, pp. 266–267.

[5] F. Heidenreich, J. Kopcsek, and C. Wende, "FeatureMapper: Mapping Features to Models," in *Proc. of ICSE'08 (Companion)*. ACM, 2008, pp. 943–944.

[6] R. Muschevici, J. Proenca, and D. Clarke, "Modular Modelling of Software Product Lines with Feature Nets," in *Proc. of SEFM'11*, 2011, pp. 318–333.

[7] I. Abal, C. Brabrand, and A. Wasowski, "42 Variability Bugs in the Linux Kernel: A Qualitative Analysis," in *Proc. of ASE'14*, 2014, pp. 421–432.

[8] T. Thüm, D. Batory, and C. Kästner, "Reasoning About Edits to Feature Models," in *Proc. of ICSE'09*. IEEE Computer Society, 2009, pp. 254–264.

[9] 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*, vol. 23, no. 4, pp. 687–733, 2016.

[10] R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik, "Lifting model transformations to product lines," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 117–128.

[11] M. Chechik, M. Famelis, R. Salay, and D. Strüber, "Perspectives of Model Transformation Reuse," in *Proc. of IFM'16*, 2016, pp. 28–44.

[12] P. Borba, L. Teixeira, and R. Gheyi, "A Theory of Software Product Line Refinement," *Theor. Comput. Sci.*, vol. 455, pp. 2–30, 2012.

[13] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza, "Safe Evolution Templates for Software Product Lines," *Journal of Systems and Software*, vol. 106, pp. 42–58, 2015.

[14] G. Sampaio, P. Borba, and L. Teixeira, "Partially Safe Evolution of Software Product Lines," in *Proc. of SPLC'16*, 2016, pp. 124–133.

[15] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer, "Generation of Visual Editors as Eclipse Plug-ins," in *Proc. of ASE '05*. ACM, 2005, pp. 134–143.

[16] D. Batory, M. Azanza, and J. Saraiva, "The objects and arrows of computational design," in *Proc. of MoDELS '08*, 2008, pp. 1–20.

[17] Z. Diskin, T. Maibaum, and K. Czarnecki, "A Model Management Imperative: Being Graphical is not Sufficient, You Have to be Categorical," in *Proc. of ECMFA'15*. Springer, 2015, pp. 154–170.

[18] F. Rabbi, Y. Lamo, and I. C. Yu, "Towards a Categorical Approach for Meta-modelling Epistemic Game Theory," in *Proc. of MoDELS'16*. ACM, 2016, pp. 57–64.

[19] S. MacLane, *Categories for the Working Mathematician*. New York: Springer-Verlag, 1971, Graduate Texts in Mathematics, Vol. 5.

[20] J. Adámek, H. Herrlich, and G. E. Strecker, *Abstract and Concrete Categories. The Joy of Cats*. Wiley-Interscience, 1990.

[21] H. Ehrig, U. Golas, A. Habel, L. Lambers, and F. Orejas, "$\mathcal{M}$-adhesive transformation systems with nested application conditions. part 1: parallelism, concurrency and amalgamation," *Mathematical Structures in Computer Science*, vol. 24, no. 4, 2014.

[22] ——, "$\mathcal{M}$-Adhesive Transformation Systems with Nested Application Conditions. Part 2: Embedding, Critical Pairs and Local Confluence," *Fundam. Inform.*, vol. 118, no. 1-2, pp. 35–63, 2012.

[23] B. Braatz, H. Ehrig, K. Gabriel, and U. Golas, "Finitary $\mathcal{M}$-adhesive categories," in *Proc. of ICGT 2010*, 2010, pp. 234–249.

[24] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, ser. Monographs in Theoretical Computer Science. Springer, 2006.

[25] S. Lack and P. Sobocinski, "Adhesive Categories," in *Proc. of FOSSACS'04*, ser. Lecture Notes in Computer Science, vol. 2987. Springer, 2004, pp. 273–288.

[26] G. Taentzer, R. Salay, D. Strüber, and M. Chechik, "Transformations of software product lines: A generalizing framework based on category theory: extended version," 2017. [Online]. Available: https://www.uni-marburg.de/fb12/arbeitsgruppen/swt/research/publications

[27] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *Proc. of SPLC'17*, 2007, pp. 23–34.

[28] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations," in *Model Driven Engineering Languages and Systems*, 2010, pp. 121–135, http://www.eclipse.org/henshin/.

[29] 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 *Proc. of ICGT'17*, 2017.

[30] S. Greiner, F. Schwägerl, and B. Westfechtel, "Realizing Multi-variant Model Transformations on Top of Reused ATL Specifications," in *Proc. of MODELSWARD'17*, 2017, pp. 326–373.

[31] K. Garcés, C. Parra, H. Arboleda, A. Yie, and R. Casallas, "Variability Management in a Model-Driven Software Product Line," *Revista Avances en Sistemas e Informática*, vol. 4, no. 2, pp. 3–12, 2007.

[32] Ø. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding Standardized Variability to Domain Specific Languages," in *Proc. of SPLC'08*, 2008, pp. 139–148.

[33] I. Schaefer, A. Worret, and A. Poetzsch-Heffter, "A model-based framework for automated product derivation," in *Proc. of MAPLE'09*, 2009, pp. 14–21.

[34] I. Schaefer and F. Damiani, "Pure delta-oriented programming," in *Proc. of FOSD'10*. ACM, 2010, pp. 49–56.

[35] D. Clarke, M. Helvensteijn, and I. Schaefer, "Abstract delta modeling," *ACM Sigplan Notices*, vol. 46, no. 2, pp. 13–22, 2011.

[36] S. Lity, M. Kowal, and I. Schaefer, "Higher-order delta modeling for software product line evolution," in *Proc. of FOSD'16*, 2016, pp. 39–48.

[37] J. Rubin and M. Chechik, "Combining Related Products into Product Lines," in *Proc. of FASE'12*, ser. LNCS, vol. 7212, 2012, pp. 285–300.

[38] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake, "Variant-Preserving Refactoring in Feature-Oriented Software Product Lines," in *Proc. of VAMOS'12*, 2012, pp. 73–81.

[39] D. Strüber, J. Rubin, M. Chechik, and G. Taentzer, "A Variability-Based Approach to Reusable and Efficient Model Transformations," in *Proc. of FASE'15*. Springer, 2015, pp. 283–298.

[40] D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger, "RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules," in *Proc. of FASE'16*. Springer, 2016, pp. 122–140.

[41] D. Strüber, "Model-driven engineering in the large: Refactoring techniques for models and model transformation systems," Ph.D. dissertation, Philipps-Universität Marburg, Germany, 2016.

[42] D. Strüber and S. Schulz, "A tool environment for managing families of model transformation rules," in *Proc. of ICGT 2017, in Memory of Hartmut Ehrig*, 2016, pp. 89–101.