Combining Related Products into Product Lines

Julia Rubin[1,2] and Marsha Chechik[1]

[1] University of Toronto, Canada
[2] IBM Research in Haifa, Israel
mjulia@il.ibm.com, chechik@cs.toronto.edu

Abstract. We address the problem of refactoring existing, closely related products into product line representations. Our approach is based on *comparing* and *matching* artifacts of these existing products and *merging* those deemed similar while explicating those that vary. Our work focuses on formal specification of a product line refactoring operator called *merge-in* that puts individual products together into product lines. We state sufficient conditions of model *compare*, *match* and *merge* operators that allow application of *merge-in*. Based on these, we formally prove correctness of the *merge-in* operator. We also demonstrate its operation on a small but realistic example.

1 Introduction

Numerous companies develop and maintain families of related software products. These products share a common, managed set of features that satisfy the specific needs of a particular market segment and are referred to as software product lines (SPLs) [4]. SPLs often emerge from experiences in successfully addressed markets with similar, yet not identical needs. It is difficult to foresee these needs a priori and hence to structure and manage the SPL development upfront [11]. As a result, SPLs are usually developed in an ad-hoc manner, using available software engineering practices such as duplication (the "clone-and-own" paradigm where artifacts are copied and modified to fit the new purpose), inheritance, source control branching and more. However, these software engineering practices do not scale well to product line development, resulting in massive rework, increased time-to-market and lost opportunities.

Software Product Line Engineering (SPLE) is a software engineering discipline aiming to provide methods for dealing with the complexity of SPL development [4,18,5]. SPLE practices promote *systematic software reuse* by identifying and managing *commonalities* – artifacts that are part of each product of the product line, and *variabilities* – artifacts that are specific to one or more (but not all) individual products across the whole product portfolio. Commonalities and variabilities are controlled by *feature models* [7] (a.k.a. *variability models*) which specify program functionality units and relationships between them. A product of the product line is identified by a unique and legal combination of features, and vice versa.

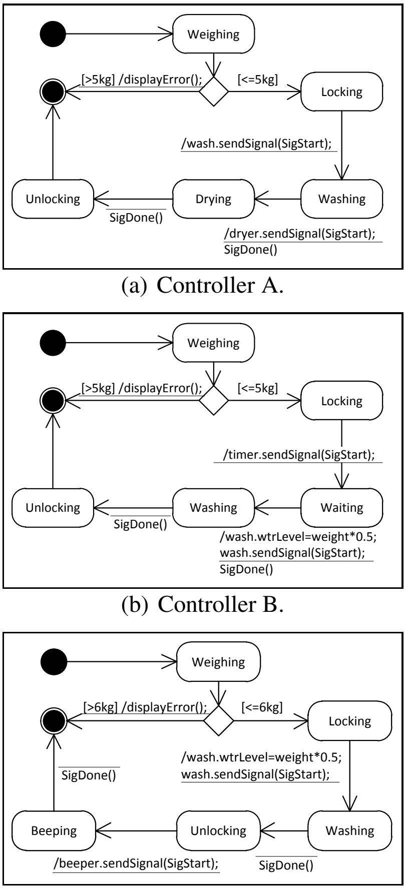
SPLE approaches can be divided into two categories: *compositional*, which implement product features as distinct fragments and allow generating specific product by composing a set of fragments, and *annotative*, which assume that there is one "maximal" product in which annotations indicate the product feature that a particular

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 285–300, 2012.

fragment realizes [8,3]. A specific product is obtained by removing fragments corresponding to discarded features. We follow the annotative approach here.

A number of works, e.g., [18,5], promote the use of annotative SPLE practices for *model-driven development* of complex systems. They are built upon the idea of explicating and parameterizing *variable* model elements by features. The parameterized elements are included in a product only if their corresponding features are selected, allowing coherent and uniform treatment of the product portfolio, a reduced number of duplications across products, better understandability and reduced maintenance effort, e.g., because modifications in the common parts can be performed only once.

Example. Consider three fragments of UML statechart controllers depicted in Fig. 1. These models were inspired by a real-life SPL developed by a partner (since partner-specific details are confidential, we move the problem into a familiar domain of washing machines). Controller A in Fig. 1(a) weighs the laundry and displays an error message if the weight is more than 5 kg. Otherwise, it locks the washing machine and sends a signal to the wash engine, responsible for performing the washing cycle. When washing is done, the Controller signals the dryer to perform the drying cycle, after which it proceeds to unlock the washing machine and finish. Controller B in Fig. 1(b) differs from the one in Fig. 1(a) by using the timer component to delay the wash cycle and by setting the wtrLevel attribute of the wash engine to the desired water level based on the weight of the laundry. This model also lacks the dryer capability. Similarly to the one in Fig. 1(b), Controller C in Fig. 1(c) uses the wtrLevel attribute to set the desired water level of the wash engine based on the laundry weight. However, it allows laundry weights up to 6 kg. It also lacks both the dryer and the timer capabilities but initiates an acoustic notification at the end of the program by invoking the beeper engine.



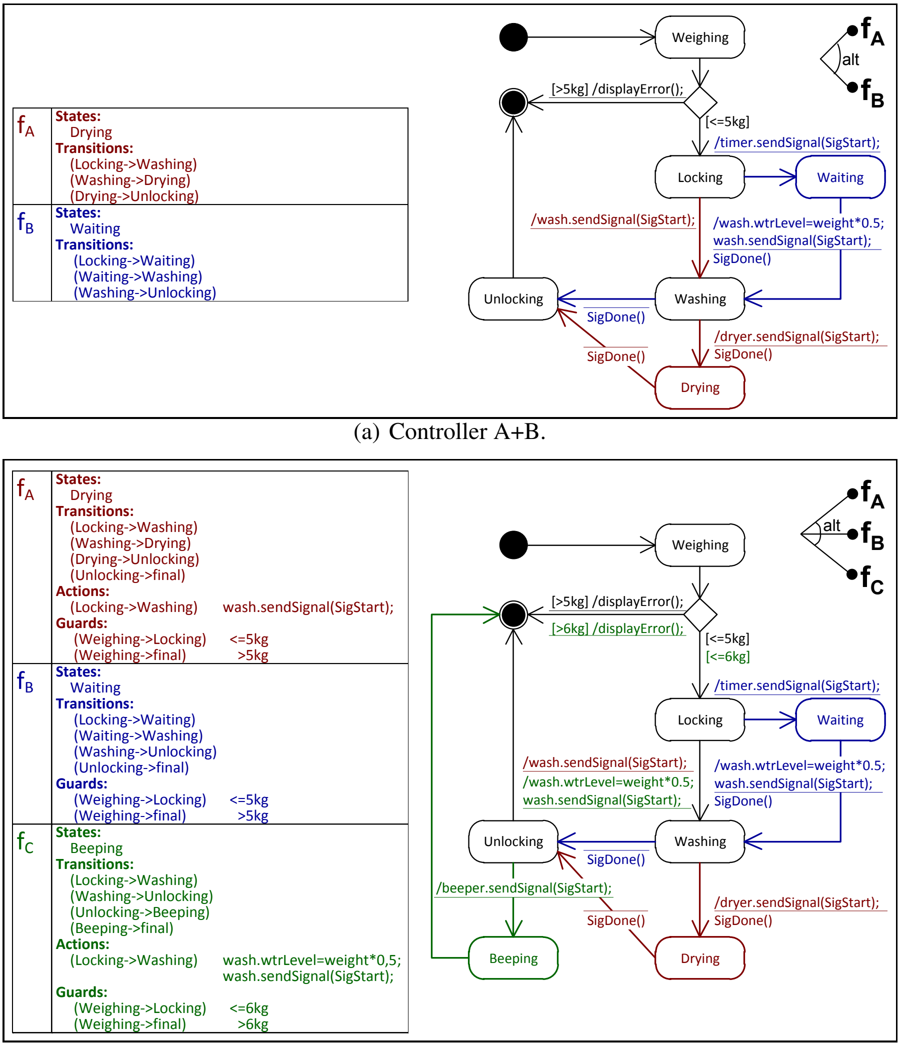
(a) Controller A.

(b) Controller B.

(c) Controller C.

Fig. 1. Washing Machine Controllers

These controllers have a large degree of similarity and can be refactored into SPLE representations where duplications are eliminated and variabilities are explicated. An example of a possible refactoring is given in Fig. 2(b), where the Drying, Waiting



(a) Controller A+B.

(b) Controller A+B+C.

Fig. 2. Possible Refactorings of the Washing Machine Controllers in Fig. 1

and `Beeping` states and their corresponding transitions are annotated by a set of features depicted in the right upper part of the figure. The refactored product line in our example encapsulates only the original input products, thus we have just three alternative features representing these products – f_A, f_B and f_C. The set of annotations specifies elements to be included given a particular feature selection. E.g., selecting f_A filters out all elements not annotated with that feature, which results in `Controller` A in Fig. 1(a). Likewise, selecting feature f_B (f_C) results in `Controller` B (`Controller` C) in Fig. 1(b) (Fig. 1(c)).

The annotations themselves are shown in a table on the left-hand side of the figure (see "State" and "Transitions" entries in the table). While the transition between Locking and Washing states exists in both Controller A and C (Fig. 1(a,c)), the corresponding actions on the transition are different and thus are also annotated by features in the combined version (see "Actions" entry in the table). Likewise, laundry weight guards on the transitions exiting the Weighing state are annotated by the corresponding features as well (see "Guards" entry in the table).

Product Line Refactoring Framework. Despite the benefits of applying SPLE practices which include improved time-to-market and quality, reduced portfolio size, engineering costs and more [4], it is impractical to assume that existing (legacy) product line systems can be abandoned altogether for creating new ones that take advantage of the SPLE reuse techniques. Thus, a transition process which involves identification and extraction of common and variable artifacts together with *variability models* that control them, becomes a necessity [12,1].

In our work, we propose a generic framework for mining legacy product lines and automating their refactoring to contemporary feature-oriented SPLE approaches, initially suggested in [19]. We consider those refactorings that just include the set of existing products rather than allowing novel feature combinations (e.g., a product with both the timer and the beeper capabilities). Our approach is based on *comparing* elements of the input products to each other (by calculating a weighted similarity of their corresponding sub-elements), *matching* those whose similarity is above a preset threshold and *merging* these together.

Our refactoring framework is applicable to a variety of model types, such as UML, EMF or Matlab/Simulink, and to different *compare*, *match* and *merge* operators. In this paper, we develop a generic model representation and a generic and parameterizable *compare / match / merge* infrastructure underlying the refactoring framework. Using them, we prove that our refactoring approach is *semantically correct*, i.e., it can generate exactly the original products, regardless of a particular implementation used and parameters chosen. The main contribution of this paper is thus the formal foundation that underlays the parameterizable and configurable, yet semantically correct refactoring framework.

There are multiple ways to *merge-in* input products into a product line, even if we only consider those refactorings that maintain the original set of input products. The resulting refactorings vary *syntactically*, depending on how elements are matched and combined. For example, in Fig. 2(b), transitions from Locking to Washing states of Controllers A and C (Fig. 1 (a,c)) are matched to each other and combined, while their corresponding actions are annotated by features. Instead, these transitions do not have to be matched, so that the generated result has two separate transitions, each annotated by the corresponding feature. Also, the Unlocking state of Controller A in Fig. 1(a) could be matched and combined with the Beeping state of Controller C in Fig. 1(c) because of their structural similarity – both transition to the final state of the statechart.

In this work, we formally prove that all these syntactically different refactorings are able to produce the set of original input products and thus are "correct". Elsewhere [20], we focus on techniques for distinguishing between multiple possible refactorings based on their qualitative properties and choosing a desired one which satisfies the set of defined objectives (e.g., one objective might be to decrease the size of the produced result, while another – to keep a low number of annotated elements per diagram). In [20], we also instantiate our approach on product lines defined in UML – a common specification language in automotive, aerospace & defense, and consumer electronics domains, and demonstrate its applicability on several large-scale examples.

The remainder of this paper is organized as follows. We introduce our data model and give the necessary background on product lines representations in Sec. 2. We give formal foundations of model merging in Sec. 3 and define our merging-based product line refactoring technique in Sec. 4. We prove semantic correctness of the technique in Sec. 5. We conclude the paper with a discussion of related work in Sec. 6, presenting a summary and future research directions in Sec. 7.

2 Preliminaries

In this section, we describe our representation of models and model elements and fix our notation for representing product line models annotated by features.

Model Representation. Following XMI principles [17], we define models to be trees of typed elements. Each element has a unique *id* which identifies it within the model and a *role* which defines the relationship between the element and its parent. For example, in UML, an element of type `Behavior` can have an `Entry` action or `Do` activity roles in a state. In addition, a single element can fulfill several roles in a model: a `Behavior` can be a `Do` activity of a state and an `Effect` of a transition at the same time. To allow reusing elements for different roles, we employ a cross-referencing mechanism where an element of type `Ref` represents the referenced element by carrying its id. Cross-referencing, combined with roles, allows representing labeled graphs using trees: an element can be linked to multiple different elements, each time in a distinct role.

Element types, denoted by $\mathbb{T}$, and roles, denoted by $\mathbb{R}$, are defined by the domain model. For UML, types include `Class`, `State`, `OpaqueBehavior`, etc. Roles include `PackagedElement`, `Subvertex`, `Effect`, etc. If the types `Ref` and `String` are not defined by the domain model, we add them to $\mathbb{T}$ as well.

We differ from [17] by representing all element attributes, as first-class model elements. That is, an element's name is represented by a separate model element of role `Name` and type `String`. The implication of our representation is that elements' attributes now have their own ids and thus, an element can have multiple attributes in the same role, e.g., multiple names or `Effects` for a transition. These qualities are required for defining the product line *merge-in* operator in Sec. 4. A formal representation of our notations is given by Def. 1 below.

Definition 1. *(Model Element) A* model element m *is a tuple* $\langle m|_{id}, m|_t, m|_r, m|_v,\ m|_s\rangle$*, where* $m|_{id}$ *is a numeric identifier of the element,* $m|_t \in \mathbb{T}$ *is the element's type,* $m|_r \in \mathbb{R}$ *is the element's role,* $m|_v$ *is the element's value – either* `String` *or an id of another element (representing a reference), and* $m|_s$ *is a (nested) list of sub-elements.*

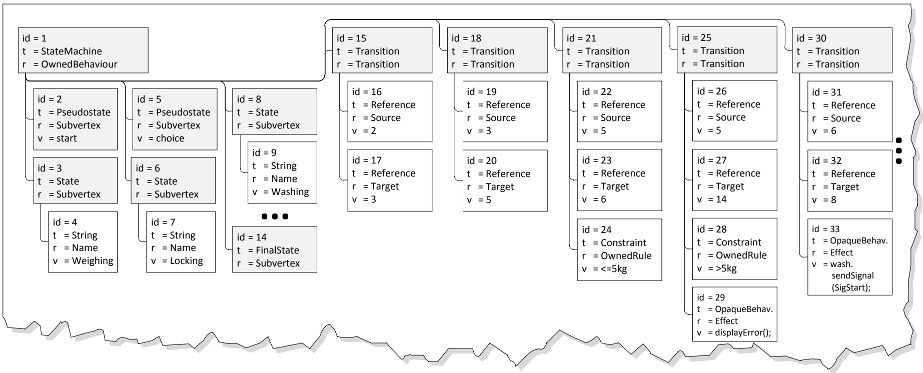

id = 1 t = StateMachine r = OwnedBehaviour
id = 2 t = Pseudostate r = Subvertex v = start
id = 3 t = State r = Subvertex
id = 4 t = String r = Name v = Weighing
id = 5 t = Pseudostate r = Subvertex v = choice
id = 6 t = State r = Subvertex
id = 7 t = String r = Name v = Locking
id = 8 t = State r = Subvertex
id = 9 t = String r = Name v = Washing
id = 14 t = FinalState r = Subvertex
id = 15 t = Transition r = Transition
id = 16 t = Reference r = Source v = 2
id = 17 t = Reference r = Target v = 3
id = 18 t = Transition r = Transition
id = 19 t = Reference r = Source v = 3
id = 20 t = Reference r = Target v = 5
id = 21 t = Transition r = Transition
id = 22 t = Reference r = Source v = 5
id = 23 t = Reference r = Target v = 6
id = 24 t = Constraint r = OwnedRule v = <=5kg
id = 25 t = Transition r = Transition
id = 26 t = Reference r = Source v = 5
id = 27 t = Reference r = Target v = 14
id = 28 t = Constraint r = OwnedRule v = >5kg
id = 29 t = OpaqueBehav. r = Effect v = displayError();
id = 30 t = Transition r = Transition
id = 31 t = Reference r = Source v = 6
id = 32 t = Reference r = Target v = 8
id = 33 t = OpaqueBehav. r = Effect v = wash. sendSignal (SigStart);

Fig. 3. Partial representation of the Statechart in Fig. 1(a)

Fig. 3 shows partial representation of the `Controller` A statechart in Fig. 1(a), where states `Drying` and `Unlocking`, together with their incoming and outgoing transitions, are omitted to save space. In this figure, sub-elements are represented as element's children in the tree.

We refer to types that have no owned properties, such as `String` or `Ref`, as *atomic*. Other types, such as `Class`, `State` or `Transition`, are *compound*. Elements of atomic and compound types are referred to as *atomic* and *compound elements*, respectively. While atomic elements have values, values of compound elements are determined from values of their sub-elements. Thus, two compound elements may be equal (i.e., have the same type and role, like elements with ids 3 and 6 in Fig. 3) but not equivalent, as they might have different sub-elements.

Definition 2. *(Equivalence) Given a universe of model elements $\mathbb{M}$, let $M_1, M_2 \in 2^{\mathbb{M}}$ be distinct sets of elements. $m_1 \in M_1$, $m_2 \in M_2$ are* equal*, denoted by $m_1 \cong m_2$, iff $m_1|_t = m_2|_t$, $m_1|_r = m_2|_r$ and $m_1|_v = m_2|_v$. Equal atomic elements are* equivalent*. Compound elements are* equivalent*, denoted by $m_1 = m_2$, iff $m_1 \cong m_2$, and their corresponding trees of sub-elements are isomorphic wrt. equality.*

Definition 3. *(Model and Model Equivalence) A set of elements $M \in 2^{\mathbb{M}}$ is a* model *iff all elements in M are connected in a tree structure by the* sub-elements *relationship, and each $m \in M$ has a unique id. Models M_1 and M_2 are* equivalent*, denoted by $M_1 = M_2$, iff their corresponding root elements are equivalent.*

Product Line Engineering. Next, we describe the formal semantics of the *annotative* SPLE approach.

Definition 4. *(Feature Model and Configuration – simplified version of [23]) Given a universe of elements $\mathbb{F}$ that represent features, a* feature model *$\mathcal{FM} = \langle \mathcal{F}, \varphi \rangle$ is a set of features $\mathcal{F} \in 2^{\mathbb{F}}$ and a propositional formula φ defined over the features from $\mathcal{F}$. A* feature configuration *$\widehat{\mathcal{FM}}$ of $\mathcal{FM}$ is a set of selected features from $\mathcal{F}$ that respect φ (i.e., φ evaluates to* true *when each variable f of φ is substituted by* true *if $f \in \widehat{\mathcal{FM}}$ and by* false *otherwise.)*

Definition 5. *(Product Line – adapted from [2]) A* product line $\mathcal{PL} = \langle \mathcal{FM}, \mathcal{M}, \mathcal{R} \rangle$ *is a triple, where* $\mathcal{FM}$ *is a feature model,* $\mathcal{M} \in 2^{\mathbb{M}}$ *is a domain model, and* $\mathcal{R} \subseteq \mathcal{F} \times \mathcal{M}$ *is a set of relationships that* annotate *elements of* $\mathcal{M}$ *by features of* $\mathcal{F}$.

Fig. 2(a) presents a snippet of a domain model, whose elements are connected to features from a feature model using annotation relationships. In this case, features f_A and f_B are alternative to each other, i.e., the propositional formula φ which specifies their relationship is $(f_A \vee f_B) \wedge \neg(f_A \wedge f_B)$. Thus, the only two valid feature configurations are $\{f_A\}$ and $\{f_B\}$.

A *specific product* derived from a product line *under a particular configuration* is a set of elements annotated by features from this configuration. For example, the statechart in Fig. 1(a) can be derived from the product line in Fig. 2(a) under the configuration $\{f_A\}$.

In this work, we assume that *common* product line elements, i.e., elements that are present in all products derived from a product line, are annotated by all features of $\mathcal{F}$. *Variable* elements are annotated by some, but not all, features of $\mathcal{F}$. To avoid clutter, we do not display annotation relationships for common product line elements in Fig. 2.

We denote by Δ the mapping between an element of the product line model and the corresponding element of the product model. We denote by Δ^{-1} the inverse mapping. For example, let m and $\hat{m}$ refer to the transition between `Locking` and `Washing` states in Fig. 1(a) and Fig. 2(a), respectively. Then, under the configuration $\{f_A\}$, $\Delta(m) = \hat{m}$ and $\Delta^{-1}(\hat{m}) = m$.

Definition 6. *(Product Derivation – adapted from [2]) Let* $\mathcal{PL} = \langle \mathcal{FM}, \mathcal{M}, \mathcal{R} \rangle$ *be a product line and let* $\widehat{\mathcal{FM}}$ *be its feature configuration. A set of model elements* $\hat{M}$ *is* derived *from the product line* $\mathcal{PL}$ under the configuration $\widehat{\mathcal{FM}}$*, denoted by* $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$*, iff the following properties hold:*

(a) An element belongs to the derived model if and only if this element is annotated by a feature of the feature configuration $\widehat{\mathcal{FM}}$ *(under which the derivation was performed):* $\forall m \in \mathcal{M}, \Delta(m) \in \hat{M} \Leftrightarrow \exists f \in \widehat{\mathcal{FM}} \cdot (f, m) \in \mathcal{R}$.

(b) Only one element can be derived from a given domain model element: $\forall m \in \mathcal{M}, \exists! \hat{m} \in \hat{M} \cdot \hat{m} = \Delta(m)$.

(c) Only derived elements are present in the derived model: $\forall \hat{m} \in \hat{M}, \exists! m \in \mathcal{M} \cdot \hat{m} = \Delta(m)$.

(d) Each element of the derived model preserves the type/role/value of its corresponding domain model element: $\hat{m} = \Delta(m) \Rightarrow \hat{m} \cong m$.

(e) Each element of the derived model preserves those sub-elements of its corresponding domain model element that were annotated by the features from $\widehat{\mathcal{FM}}$*:* $\forall \hat{m} \in \hat{M}$, $\hat{m}^c \in \hat{m}|_s \Leftrightarrow \Delta^{-1}(\hat{m}^c) \in \Delta^{-1}(\hat{m})|_s \wedge \exists f \in \widehat{\mathcal{FM}} \cdot (f, \Delta^{-1}(\hat{m}^c)) \in \mathcal{R})$.

It is easy to show that a feature model configuration uniquely identifies the derived product model.

Lemma 1. *(Uniqueness) Let* $\mathcal{PL} = \langle \mathcal{FM}, \mathcal{M}, \mathcal{R} \rangle$ *be a product line,* $\widehat{\mathcal{FM}}$ *be a feature configuration and* $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$*. Then, for each* $\hat{M}' = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$, $\hat{M}' = \hat{M}$.

Table 1. State Similarity Weights $\mathbb{W}$ Used by *Compare* for Fig. 1

Element	Name	Type	Depth	Actions	Transitions
Weight	0.2	0.05	0.1	0.3	0.35

Proof. Assume to the contrary that $\hat{M}' \neq \hat{M}$ and assume without loss of generality that $\exists \hat{m} \in \hat{M}$ such that $\hat{m} \notin \hat{M}'$. By Def. 6(c), $\hat{m} \in \hat{M}$ implies that $\exists m \in \mathcal{M} \cdot \hat{m} = \Delta(m)$. By Def. 6(a), this means that $\exists f \in \widehat{\mathcal{FM}} \cdot (f, m) \in \mathcal{R}$. Since $\hat{M}'$ was derived from $\mathcal{PL}$ under the same configuration $\widehat{\mathcal{FM}}$, $\Delta(m) \in \hat{M}'$ by Def. 6(a), which implies that $\exists \hat{m}' \in \hat{M}' \cdot \hat{m}' = \Delta(m)$ by Def. 6(b). Since $\hat{m} = \Delta(m) = \hat{m}'$, we conclude that $\hat{m} \in \hat{M}'$ which creates a contradiction.

3 Model Merging

In this section, we formalize properties of *model merging* [22,16]. Model merging is an operation which consists of (1) *compare*, which determines how similar model elements are to each other, (2) *match*, which detects pairs of elements that should constitute a match and (3) *merge*, which puts information contained in input models together while keeping a single copy of matched elements. We specify the minimal set of properties that these three *model merging* steps should satisfy in order to be used for combining individual products into product lines.

Compare is a heuristic function that calculates the similarity degree, a number between 0 and 1, for each pair of input model elements. It receives models M_1, M_2 and a set of empirically computed weights $\mathbb{W} = \{w_R \mid R \in \mathbb{R}\}$ which represent the contribution of sub-elements in role R to the overall similarity of their owning elements.

For the example in Fig. 1, a similarity degree between two states is calculated as a weighted sum of the similarity degrees of their names, entry and exit actions, do activities, incoming and outgoing transitions, etc.[1] Comparing `Locking` states from Fig. 1(a,b) to each other yields a relatively high similarity degree of 0.85, as these elements have identical names and similar incoming transitions. However, their outgoing transitions have different actions and lead to non-similar states; thus, the states are not identical. Comparing `Drying` and `Waiting` states yields a lower number, as these states have different names and different incoming and outgoing transitions.

Definition 7. *(*Compare*) Let $M_1, M_2 \in 2^{\mathbb{M}}$ be models.* Compare$(M_1, M_2, \mathbb{W})$ *is a total function that produces a set of triples $C \subseteq (M_1 \times M_2 \times [0..1])$ that satisfy the following properties:*

(a) The similarity degree of equal elements is 1*:* $(m_1 = m_2) \Rightarrow (m_1, m_2, 1) \in C$.

(b) The similarity degree of elements having different types or roles is 0*:*
$(m_1|_t \neq m_2|_t) \vee (m_1|_r \neq m_2|_r) \Rightarrow (m_1, m_2, 0) \in C$.

(c) While comparing, references are substituted by the elements they refer to:
$m_1|_t = m_2|_t = \texttt{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (M_1[m_1|_v], M_2[m_2|_v], x) \in C)$;
$m_1|_t = \texttt{Ref} \wedge m_2|_t \neq \texttt{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (M_1[m_1|_v], m_2, x) \in C)$;
$m_1|_t \neq \texttt{Ref} \wedge m_2|_t = \texttt{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (m_1, M_2[m_2|_v], x) \in C)$.

[1] Some *compare* algorithms, e.g., [16], might perform several iterations until they stabilize and calculate the final similarity degree between elements.

(d) $\text{compare}_{T,R}$ *are domain-specific functions, used to calculate the similarity degree between atomic elements of type* T *in role* R *(e.g., elements' names):* $m_1|_t = m_2|_t = T$, $m_1|_r = m_2|_r = R$, T *is atomic* $\Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow x = \text{compare}_{T,R}(m_1, m_2))$.

(e) *The similarity degree of compound elements is calculated as a weighted sum of their sub-elements' similarity:* $m_1|_t = m_2|_t = T$, T *is compound* $\Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow x = \sum_{\{R\}} w_R * s_R)$*, where* $\{R\}$ *is a set of possible roles for sub-elements of* T*,* w_R *is the contribution of sub-elements in role* R *to the overall similarity of* T *(* $\sum_{\{R\}} w_R = 1$*), and* s_R *is the calculated similarity between sub-elements of* m_1 *and* m_2 *in role* R*.*

Modifying weights $\mathbb{W}$ can produce syntactically different matches. To obtain the model in Fig. 2(b), we calculated state similarity using weights in Table 1, which were set empirically. Decreasing the weight of the name similarity between states while increasing the weight of the similarity of their corresponding incoming and outgoing transitions could, for example, result in lowering the similarity degree between `Washing` states in Fig. 1(a,c) from 0.8 to 0.7, as their incoming and outgoing transitions differ significantly. This can subsequently lead to not matching these states and thus, unlike in the model in Fig. 2(b), each would be present in the resulting refactoring.

Match is a heuristic function that receives pairs of model elements together with their similarity degree and returns those pairs that are considered similar, using empirically determined *similarity thresholds* $\mathbb{S} = \{S_T \mid T \in \mathbb{T}\}$. Matched elements are combined together by the *merge* function, while unmatched are copied to the result without modification.

Definition 8. *(*Match*) Let* $M_1, M_2 \in 2^{\mathbb{M}}$ *be models and let* C *be a set of triples produced by* $\text{compare}(M_1, M_2, \mathbb{W})$*. Then,* $\text{match}(M_1, M_2, C, \mathbb{S})$ *is a function that produces a set of pairs* $S \subseteq (M_1 \times M_2)$ *that satisfy the following properties:*

(a) *Each element from* M_1 *can be matched with only one element of* M_2*, and vice versa:* $(m_1, m_2) \in S \Rightarrow \forall (m_1', m_2') \in S (m_1'|_{id} = m_1|_{id} \Leftrightarrow m_2'|_{id} = m_2|_{id})$*.*

(b) *Only identical atomic elements are matched:*
$m_1|_t = m_2|_t = T$, T *is atomic* $\Rightarrow (m_1, m_2) \in S \Leftrightarrow (m_1, m_2, 1) \in C$.

(c) *Compound elements are matched only if their similarity degree exceeds the threshold that is set for their type:*
$m_1|_t = m_2|_t = T$, T *is compound* $\Rightarrow (m_1, m_2) \in S \Leftrightarrow (m_1, m_2, x) \in C \land x \geq S_T$.

(d) *If two elements are matched, their parent elements are matched as well (e.g., it is not possible to match transition guards without matching the owning transitions):* $(m_1, m_2) \in S \Rightarrow (\exists m_1^p \in M_1, m_2^p \in M_2 \cdot m_1 \in m_1^p|_s \land m_2 \in m_2^p|_s \Rightarrow (m_1^p, m_2^p) \in S)$.

(e) *Either root elements of* M_1 *and* M_2 *are matched with each other, or one of them has no match at all:* $\neg\exists m_1^p \in M_1 \cdot m_1 \in m_1^p|_s \land \neg\exists m_2^p \in M_2 \cdot m_2 \in m_2^p|_s \Rightarrow ((m_1, m_2) \in S \lor \neg\exists m_1' \in M_1 \cdot (m_1', m_2) \in S \lor \neg\exists m_2' \in M_2 \cdot (m_1, m_2') \in S)$.

Consider the above example where `Washing` states had the calculated similarity degree of 0.8 and 0.7 for two different settings of *compare* weights $\mathbb{W}$. Setting the state similarity threshold to 0.75 results in matching the states to each other in the former case and not matching in the latter. Likewise, the transitions between `Locking` and `Washing` states in Fig. 1(a,c) can be matched, resulting in the refactoring in Fig. 2(b), where the corresponding actions are parameterized by features, or not matched, resulting in two separate parameterized transitions.

Merge is a function that receives two models together with pairs of their matched elements and returns a merged model that contains all elements of the input, while matched elements are unified and appear in the resulting model only once.

We denote by σ the mapping from an element of an input model to its corresponding element in the merged result, and say that σ *transforms* an input model element to its corresponding element in the result. We denote by σ_1^{-1} and σ_2^{-1} the reverse mappings from an element in the merged result to its *origin* in the first and second models, respectively (or $\emptyset$ if such an element does not exist in one of them). For example, let m_1, m_2 and m denote the states `Washing` in the models in Fig. 1(a), 1(b) and 2(a), respectively. Then, $\sigma(m_1) = \sigma(m_2) = m$, $\sigma_1^{-1}(m) = m_1$ and $\sigma_2^{-1}(m) = m_2$.

Definition 9. *(*Merge*) Let $M_1, M_2 \in 2^{\mathbb{M}}$ be models, C be a set of triples produced by* compare($M_1, M_2, \mathbb{W}$) *and S be a set of pairs produced by* match($M_1, M_2, C, \mathbb{S}$)*. Then,* merge(M_1, M_2, S) *is a function that produces the merged model $\bar{M}$ and satisfies the following properties:*

(a) Matched elements are transformed to the same element in the output model $\bar{M}$: $(m_1, m_2) \in S \Leftrightarrow \sigma(m_1) = \sigma(m_2)$.

(b) Each input model element is transformed to exactly one element of $\bar{M}$: $\forall m_1 \in M_1, \exists! \bar{m} \in \bar{M} \cdot \bar{m} = \sigma(m_1)$ *and* $\forall m_2 \in M_2, \exists! \bar{m} \in \bar{M} \cdot \bar{m} = \sigma(m_2)$.

(c) Each element of $\bar{M}$ is created from an element of M_1 and/or an element of M_2. Moreover, no two distinct elements of an input model can be transformed to the same element in the result: $\forall \bar{m} \in \bar{M} \cdot (\exists! m_1 \in M_1 \cdot m_1 = \sigma_1^{-1}(\bar{m})) \vee (\exists! m_2 \in M_2 \cdot m_2 = \sigma_2^{-1}(\bar{m}))$.

(d) Each element of $\bar{M}$ preserves the type, role and value of its corresponding original elements. (By Def. 7(b) and 8(b), only elements with the same type, role and value can be matched: atomic elements are matched only if identical, while compound elements do not have values.) $\forall m \in M_1 \cup M_2, \forall \bar{m} \in \bar{M}, \bar{m} = \sigma(m) \Rightarrow \bar{m} \cong m$.

(e) Each element of $\bar{M}$ preserves sub-elements of its corresponding original elements: $\forall \bar{m} \in \bar{M}, \bar{m}^c \in \bar{m}|_s \Leftrightarrow (\sigma_1^{-1}(\bar{m}^c) \in \sigma_1^{-1}(\bar{m})|_s) \vee (\sigma_2^{-1}(\bar{m}^c) \in \sigma_2^{-1}(\bar{m})|_s)$*.*

While the *compare* and *match* functions rely on heuristically set weights $\mathbb{W}$ and similarity degrees $\mathbb{S}$, *merge* is not heuristic: its output is uniquely defined by the input set of matched elements. For this work, we rely on *union-merge* [22] realization of the *merge* function. *Union-merge* unifies matched elements and copies unmatched elements "as is" to the result. Since our data model in Sec. 2 represents attributes of model elements as separate entities, an element in the merged result can have several attributes of the same type fulfilling the same role (which, for example, is not allowed by UML for effects on a transition or state *do* activities). We use this property of the data model to capture annotative product line representations generated when merging individual products into product lines.

4 Product Line Refactoring

In this section, we define the *merge-in* operator, which is used to put together input products into a product line. It constructs a product line by adding input products one by one and has two parameters: an (already constructed) product line and the next model to add[2]. For the example in Fig 1, combining `Controller` A and B in Fig. 1(a,b) results

[2] The first product is implicitly converted into a "primitive" product line – a product line with only one feature and a set of annotations that relate all model elements to that feature.

in a product line A+B depicted in Fig. 2(a), with features f_A and f_B. Selecting the first one derives the original statechart of `Controller` A, while selecting the second – that of `Controller` B. Subsequent *merge-in* of `Controller` C (Fig. 1(c)) into this product line produces a representation depicted in Fig. 2(b), out of which all three original statecharts can be derived.

Definition 10. *(*Merge-in *Construction)* $\mathcal{PL}' = \langle \mathcal{FM}', \mathcal{M}', \mathcal{R}' \rangle$ *is a product line constructed by* merging-in *a product* M *into the product line* $\mathcal{PL}$ *(denoted by* $\mathcal{PL}' = \mathcal{PL} \oplus_{\mathbb{W},\mathbb{S}} M$*), using the rules below:*

(a) A new feature f_M*, representing the* merged-in *product* M*, is added as an alternative to all existing features: if* $\mathcal{FM} = \langle \mathcal{F}, \varphi \rangle$ *then* $\mathcal{FM}' = \langle \mathcal{F}', \varphi' \rangle$, $\mathcal{F}' = \mathcal{F} \cup \{f_M | f_M \in \mathbb{F}, f_M \notin \mathcal{F}\}$*, and* $\varphi' = (\varphi \vee f_M) \wedge \bigwedge_{f \in \mathcal{F}} \neg(f_M \wedge f)$*.*

(b) The domain model is generated by merging *the existing domain model with the newly added model* M*: if* $C = \text{compare}(\mathcal{M}, M, \mathbb{W})$ *and* $S = \text{match}(\mathcal{M}, M, C, \mathbb{S})$ *then* $\mathcal{M}' = \text{merge}(\mathcal{M}, M, S)$*.*

(c) The set of annotation relationships is enhanced by the relationships that annotate elements that originated in M *by* f_M*:* $\mathcal{R}' = \{(f, \sigma(m)) \mid f \in \mathcal{F}, m \in \mathcal{M}, (f, m) \in \mathcal{R}\} \cup \{(f_M, \sigma(m)) \mid m \in M\}$*.*

We refer to $\mathcal{PL}$ *as the* original *product line and to* $\mathcal{PL}'$ *as the* constructed *product line.*

5 Correctness of Product Line Refactoring

In this section, we prove the correctness of the *merge-in* operator introduced in Sec. 4. Specifically, we show that *merge-in* produces minimal behavior-preserving product line refinements [2], that is, the input product models are the only ones which can be derived from the refactored product line model (Theorem 1).

In what follows, let $\mathbb{W}$ be a set of weights used by the *compare* function and $\mathbb{S}$ be a set of similarity thresholds used by the *match* functions. Let $\mathcal{PL} = \langle \mathcal{FM}, \mathcal{M}, \mathcal{R} \rangle$ be a product line.

***Merge-in* Monotonicity.** Lemma 2 below shows that any feature configuration that contains only features from the original product line $\mathcal{PL}$ is also a valid feature configuration for the constructed product line $\mathcal{PL}'$, i.e., it complies to the constrains φ defined on the features of $\mathcal{PL}'$. For the example in Fig. 2, this means that a feature configuration of the product line A+B in Fig. 2(a), e.g., $\{f_A\}$, is also a valid feature configuration for the "extended" product line A+B+C in Fig. 2(b).

Lemma 2. *Let* $\widehat{\mathcal{FM}}$ *be a subset of* $\mathcal{FM}$*. Then,* $\widehat{\mathcal{FM}}$ *is a feature configuration of* $\mathcal{FM}$ *if and only if it is a feature configuration of* $\mathcal{FM}'$*.*

Proof. By construction of φ' (Def. 10(a)), $\varphi' = (\varphi \vee f_M) \wedge \bigwedge_{f \in \mathcal{F}} \neg(f_M \wedge f)$. Since $f_M \notin \widehat{\mathcal{FM}}$, $\neg(f_M \wedge f)$ evaluates to *true* for every f, and $\varphi' = (\varphi \vee \mathit{false}) = \varphi$. Thus, $\widehat{\mathcal{FM}}$ respects φ if and only if it respects φ'.

Lemma 3 shows that, under configurations used in Lemma 2, a model derived from $\mathcal{PL}$ is equal to the one derived from $\mathcal{PL}'$. That is, under the configuration $\{f_A\}$, the same model of `ControllerA` in Fig. 1(a) is derived from both product lines A+B and A+B+C (Fig. 2(a) and (b), respectively).

Lemma 3. *Let $\widehat{\mathcal{FM}}$ be a subset of $\mathcal{FM}$. If $\widehat{\mathcal{FM}}$ is a feature configuration for $\mathcal{FM}$, $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$ and $\hat{M}' = \Delta(\mathcal{PL}', \widehat{\mathcal{FM}})$, then $\hat{M} = \hat{M}'$. That is, given a feature configuration that contains only features from $\mathcal{PL}$, a set of elements that is generated from $\mathcal{PL}$ is equivalent to that generated from $\mathcal{PL}'$, under the same configuration.*

Proof. To prove the lemma, we show that $f = \Delta(\sigma(\Delta^{-1}(.)))$ is an isomorphism between the elements of $\hat{M}$ and the elements of $\hat{M}'$ that respects $\cong$. That is, we prove the following four statements, showing that f is an edge-preserving bijection. The construction of the corresponding elements in $\hat{M}$ and $\hat{M}'$ is schematically sketched in Fig. 4.

1. Any element of $\hat{M}$ has the corresponding equal element in $\hat{M}'$: $\forall \hat{m} \in \hat{M}, \exists! \hat{m}' \in \hat{M}' \cdot \hat{m}' = f(\hat{m}) \wedge \hat{m}' \cong \hat{m}$.
Let $\hat{m} \in \hat{M}$. By Def. 6(a), this means that there exists an element $m \in \mathcal{M}$, and a feature $f \in \mathcal{FM}$, such that $(f, m) \in \mathcal{R}$ and $\hat{m} = \Delta(m)$. By Def. 9(b), m is transformed by *merge* to an element $\bar{m}' \in \mathcal{M}'$, such that $\bar{m}' = \sigma(m)$. By Def. 10(c), this element is annotated by the same feature as m: $(f, \sigma(m)) \in \mathcal{R}'$. Thus, $\Delta(\sigma(m)) \in \hat{M}$ by Def. 6(a). Since $\hat{m}$ is derived from m, $m = \Delta^{-1}(\hat{m})$. It follows that $\Delta(\sigma(\Delta^{-1}(\hat{m}))) \in \hat{M}'$. Let's denote that element by $\hat{m}'$. There exists only one such element by Def. 6(b,c) and 9(b). $\hat{m}' \cong \hat{m}$ by Def. 6(d) and 9(d).

2. Any element of $\hat{M}'$ has the corresponding equal element in $\hat{M}$: $\forall \hat{m}' \in \hat{M}', \exists! \hat{m} \in \hat{M} \cdot \hat{m}' = f(\hat{m}) \wedge \hat{m}' \cong \hat{m}$.
Let $\hat{m}' \in \hat{M}'$. By Def. 6(a), this means that there exist an element $\bar{m}' \in \mathcal{M}'$, and a feature $f \in \mathcal{FM}$, such that $(f, \bar{m}') \in \mathcal{R}'$ and $\hat{m}' = \Delta(\bar{m}')$. By Def. 9(c), there are three possible cases: (1) $\sigma_1^{-1}(\bar{m}') \in \mathcal{M}, \sigma_2^{-1}(\bar{m}') = \emptyset$; (2) $\sigma_1^{-1}(\bar{m}') = \emptyset, \sigma_2^{-1}(\bar{m}') \in M$; (3) $\sigma_1^{-1}(\bar{m}') \in \mathcal{M}$, $\sigma_2^{-1}(\bar{m}') \in M$.
For cases (1) and (3), $(f, \bar{m}') \in \mathcal{R}'$ implies that $(f, \sigma_1^{-1}(\bar{m}')) \in \mathcal{R}$ by Def. 10(c), and thus, $\Delta(\sigma_1^{-1}(\bar{m}')) \in \hat{M}$ by Def. 6(a). Let's denote this element by $\hat{m}$. It is easy to see that $f(\hat{m}) = \hat{m}'$ (that is $\Delta(\sigma(\Delta^{-1}(\hat{m}))) = \hat{m}'$. There exists only one such element $\hat{m}$ by Def. 6(b,c) and 9(c). $\hat{m}' \cong \hat{m}$ by Def. 6(d) and 9(d). For case (2), $\sigma_1^{-1}(\bar{m}') = \emptyset$ implies by Def. 10(c), that $\bar{m}'$ is annotated by f_M, and, since $f_M \notin \widehat{\mathcal{FM}}$, $\Delta(\bar{m}') \notin \hat{M}'$, which, together with $\hat{m}' = \Delta(\bar{m}')$, creates a contradiction to $\hat{m}' \in \hat{M}'$.

3. Any sub-element of $\hat{m}$ has the corresponding sub-element in $f(\hat{m})$: $\forall \hat{m} \in \hat{M}(\hat{m}^c \in \hat{m}|_s \Rightarrow f(\hat{m}^c) \in f(\hat{m})|_s)$.
Since $\hat{m}^c \in \hat{m}|_s$, by Def. 6(a,e), there exist elements $m, m^c \in \mathcal{M}$, and features $f, f^c \in \mathcal{FM}$, such that $(f, m) \in \mathcal{R}$, $(f^c, m^c) \in \mathcal{R}$, $\hat{m} = \Delta(m)$, $\hat{m}^c = \Delta(m^c)$ and $m^c \in m|_s$ (it is also possible that $f = f^c$). By Def. 9(b,e), $\sigma(m^c) \in \sigma(m)|_s$. By Def. 10(c), $(f, \sigma(m)) \in \mathcal{R}'$ and $(f^c, \sigma(m^c)) \in \mathcal{R}'$, which, by Def. 6(a,e), implies that $\Delta(\sigma(m^c)) \in \Delta(\sigma(m))|_s$. Since $m^c = \Delta^{-1}(\hat{m}^c)$ and $m = \Delta^{-1}(\hat{m})$, $f(\hat{m}^c) \in f(\hat{m})|_s)$, as desired.

4. Any sub-element of $\hat{m}'$ has the corresponding sub-element in $\hat{m}$: $\forall \hat{m}' \in \hat{M}'(\hat{m}'^c \in \hat{m}'|_s \Rightarrow \exists \hat{m}, \hat{m}^c \in \hat{M} \cdot \hat{m}' = f(\hat{m}) \wedge \hat{m}'^c = f(\hat{m}^c) \wedge \hat{m}^c \in \hat{m}|_s$.
Let $\hat{m}'^c, \hat{m}' \in \hat{M}'$ be elements such that $\hat{m}'^c \in \hat{m}'|_s$. By Def. 6(a,e), there exist elements $\bar{m}', \bar{m}'^c \in \mathcal{M}'$, and features $f, f^c \in \mathcal{FM}$, such that $(f, \bar{m}') \in \mathcal{R}'$, $(f^c, \bar{m}'^c) \in \mathcal{R}'$, $\hat{m}' = \Delta(\bar{m}')$, $\hat{m}'^c = \Delta(\bar{m}'^c)$ and $\bar{m}'^c \in \bar{m}'|_s$ (it is also possible that $f = f^c$). Similarly to case 2, $\sigma_1^{-1}(\bar{m}') \neq \emptyset$ and $\sigma_1^{-1}(\bar{m}'^c) \neq \emptyset$. By Def. 9(e), either $\sigma_1^{-1}(\bar{m}'^c) \in \sigma_1^{-1}(\bar{m}')|_s$ or there exist $m_1, m_2 \in M$, such that $\sigma_1^{-1}(\bar{m}'^c)$ is matched with m_1, $\sigma_1^{-1}(\bar{m}')$ is matched with m_2, and $m_1 \in m_2|_s$. The later case is impossible by Def. 8(a,d,e) – we omit the details due to the space limitations. For the former case, since $(f, \bar{m}') \in \mathcal{R}'$, $(f^c, \bar{m}'^c) \in \mathcal{R}'$, by Def. 10(c), $(f, \sigma_1^{-1}(\bar{m}')) \in$

$\mathcal{R}$, $(f^c, \sigma_1^{-1}(\bar{m}'^c)) \in \mathcal{R}$ and thus, by Def. 6(a,e), $\Delta(\sigma_1^{-1}(\bar{m}'^c))) \in \Delta(\sigma_1^{-1}(\bar{m}')))|_s$. Let's denote these elements by $\hat{m}^c$ and $\hat{m}$, respectively. $f(\hat{m}^c)) = \Delta(\bar{m}'^c) = \hat{m}'^c$ and $f(\hat{m})) = \Delta(\bar{m}') = \hat{m}'$, implies $\hat{m}^c \in \hat{m}|_s$, as desired.

The above lemma implies that our construction preserves the behavior of the original product line model: the set of models derived from $\mathcal{PL}$ can still be derived from $\mathcal{PL}'$, as shown by the following corollary.

Corollary 1. *Let $\lfloor \mathcal{PL} \rfloor$ denote a set of all models derived from a product line $\mathcal{PL}$. That is, $\lfloor \mathcal{PL} \rfloor = \{\Delta(\mathcal{PL}, \widehat{\mathcal{FM}}) \mid \widehat{\mathcal{FM}}$ is a feature configuration of $\mathcal{FM}\}$. Then, a set of models derived from $\mathcal{PL}$ can be derived from $\mathcal{PL}'$ as well: $\lfloor \mathcal{PL} \rfloor \subseteq \lfloor \mathcal{PL}' \rfloor$.*

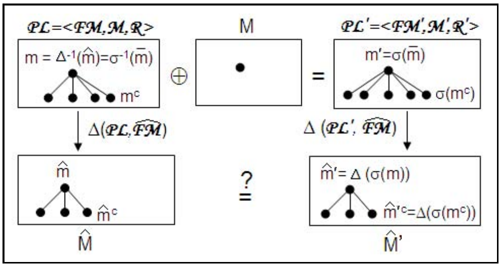


Fig. 4. A sketch for the proof of Lemma 3

Proof. For each $\hat{M} \in \lfloor \mathcal{PL} \rfloor$, there exists a configuration $\widehat{\mathcal{FM}}$, such that $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}})$. By Lemmas 2 and 3, $\hat{M} = \Delta(\mathcal{PL}', \widehat{\mathcal{FM}})$. Thus, $\hat{M} \in \lfloor \mathcal{PL}' \rfloor$.

For the example in Fig. 2, the above corollary means that both `Controller` A and `Controller` B that can be derived from the product line A+B in Fig. 2(a) can still be derived from the constructed product line A+B+C in Fig. 2(b), after `Controller` C was *merged-in* to it.

***Merge-in* Behavior Preservation.** We now show that model M which we *merge-in* into the original product line $\mathcal{PL}$ can be derived from the constructed product line $\mathcal{PL}'$. That is, when we *merge-in* `Controller` C in Fig. 1(c) into the product line A+B in Fig. 2(a), we can derive it back from the constructed product line A+B+C in Fig. 2(b).

Since f_M is the feature that annotates elements of the *merged-in* model (f_C in our example), we first show that $\{f_M\}$ is a valid feature configuration (Lemma 4). Then, Lemma 5 shows that the original model M is derived from the constructed product line $\mathcal{PL}'$ under that configuration.

Lemma 4. *$\{f_M\}$ is a feature configuration for $\mathcal{PL}'$.*

Proof. By construction of $\mathcal{FM}'$ (Def. 10(a)), $f_M \in \mathcal{F}'$. We now show that $\{f_M\}$ respects $\varphi' = (\varphi \vee f_M) \wedge \bigwedge_{f \in \mathcal{F}} \neg(f_M \wedge f)$. Since $f \notin \{f_M\}$ for any $f \in \mathcal{F}$, $\neg(f_M \wedge f)$ evaluates to *true* for every $f \in \mathcal{F}$. Since $f_M =$ *true*, $\varphi \vee f_M$ also evaluated to *true*. It follows that $\{f_M\}$ respects φ' and is a feature configuration for $\mathcal{PL}'$.

Lemma 5. *Let $\{f_M\}$ be a feature configuration. Then, a model that is derived from $\mathcal{PL}'$ under that configuration is equivalent to M. That is, $M = \Delta(\mathcal{PL}', \{f_M\})$.*

The proof of this lemma, similarly to the proof of Lemma 3, shows that $f = \sigma(\Delta(.))$ is an isomorphism between the elements of M and the elements of $\hat{M}'$, and is omitted.

Finally, Theorem 1 shows that our *merge-in* operator is behavior preserving: the set of product models that are derived from the constructed product line $\mathcal{PL}'$ is equal to the set of models that are derived from the original product line $\mathcal{PL}'$, in addition to the *merged-in* model M.

Theorem 1. $\lfloor \mathcal{PL}' \rfloor = \lfloor \mathcal{PL} \rfloor \cup \{M\}$.

Proof. We first prove that $\lfloor \mathcal{PL}' \rfloor \subseteq \lfloor \mathcal{PL} \rfloor \cup \{M\}$. Let $\hat{M} \in \lfloor \mathcal{PL}' \rfloor$ be a model derived from $\mathcal{PL}'$. Then there exists a feature configuration $\widehat{\mathcal{FM}}'$, such that $\hat{M} = \Delta(\mathcal{PL}', \widehat{\mathcal{FM}}')$. Let f_M be in $\mathcal{FM}' \smallsetminus \mathcal{FM}$.

1. If $f_M \notin \widehat{\mathcal{FM}}'$, then $\widehat{\mathcal{FM}}' \subseteq \mathcal{FM}$. Thus, by Lemma 3, $\hat{M} = \Delta(\mathcal{PL}, \widehat{\mathcal{FM}}')$, which implies that $\hat{M} \in \lfloor \mathcal{PL} \rfloor$.
2. If $f_M \in \widehat{\mathcal{FM}}'$, then, by construction of $\mathcal{FM}'$ (Def. 10(a)), $\widehat{\mathcal{FM}}' = \{f_M\}$. By Lemma 5, $M = \Delta(\mathcal{PL}', \widehat{\mathcal{FM}}')$. Thus, by Lemma 1, $\hat{M} = M$.

We now show that $\lfloor \mathcal{PL} \rfloor \cup \{M\} \subseteq \lfloor \mathcal{PL}' \rfloor$. $\lfloor \mathcal{PL} \rfloor \subseteq \lfloor \mathcal{PL}' \rfloor$ by Corollary 1. By the construction of $\mathcal{FM}'$ (Def. 10(a)), $f_M \in \mathcal{F}'$. Thus, by Lemmas 4 and 5, $\{f_M\}$ is a valid feature configuration for $\mathcal{PL}'$ and $M = \Delta(\mathcal{PL}', \{f_M\})$, which implies that $M \in \lfloor \mathcal{PL}' \rfloor$.

For the example in Fig. 2, where `Controller` C in Fig. 1(c) is *merged-in* into the product line A+B containing `Controller` A and B, this means that `Controller` A, B, and C, and only them, can be derived from the constructed product line A+B+C in Fig. 2(b).

6 Related Work

A general theory of product line refinement was introduced in [2] where the authors established product line properties supporting stepwise and compositional product line development and evolution. Our approach instantiates this theory by providing a concrete refactoring technique for combining products into product lines. We prove that our refactoring is the minimal behavior-preserving product line refinement, according to the definition in [2].

Several works (e.g., [9,10]) capture guidelines and techniques for manually transforming legacy product line artifacts into SPLE representations. Instead, our goal is to introduce automation into the refactoring process by *comparing*, *matching* and *merging* artifacts to each other. While no automated approach can replace a human product line designer and produce a solution which is as good as a hand-crafted one, automation can assist the designer and speed-up the refactoring process.

Similarly to us, Koschke et. al. [11] and Ryssel et. al. [21] introduce automatic approaches to re-organize product variants into annotative representations while identifying variation points and their dependencies. The former work reasons about components, interfaces and their grouping into subsystems. The latter works on Matlab models. Our work differs from both [11] and [21] by exploring product line commonalities and variabilities for any type of model that can be represented as XMI and by providing a formal proof of correctness of our approach.

Feature-oriented refactoring [13,15] focuses on identifying the code for a feature and factoring the code out into a single module or aspect aiming at decomposing a program into features. Since our aim is consolidation of variants into single-base product line representations, these are out of the scope for our work. Similarly, UML model refactoring (e.g., [6,24]) and code refactoring techniques (e.g., [14]), while closely related to our work, usually focus on improving the internal structure and design of a software system rather than on identifying and restructuring the system's common and variable parts.

7 Conclusion and Future Work

Extracting product line representations from existing legacy product line systems can support product line engineering adoption: reusing and leveraging knowledge accumulated in the legacy systems during their development lifetime can be more efficient than "starting from scratch". In this work, we formally specified a simple data model and a refactoring technique for transforming individual products into more compact product line representations. Our data model, inspired by XMI principles, is powerful enough to accommodate labeled-graph representations, in particular, UML. At the same time, it is flexible enough to support product line notations where several alternative elements can fulfill the same role, which is not allowed by UML itself.

Relying on the data model, we formally stated necessary and sufficient conditions allowing us to use model *compare*, *match* and *merge* operators for combining individual products into product lines. We proved that once these conditions are satisfied, the *merge-in* can be safely applied for combining products into product lines, as it produces representations that encode precisely the set of initial products. This provides formal foundation that underlays the parameterizable and configurable, yet semantically correct refactoring framework. The applicability of the framework to real-life examples, as well as techniques for distinguishing between different possible refactorings, is studied elsewhere [20].

There are several directions for continuing this work. First, we are interested in exploring more sophisticated refactoring techniques that are able to detect fine-grained features in the combined products. This would allow us to create new products in the product line by "mixing" features from different original products. We also plan to enhance *model merging* techniques with additional capabilities, such as using code-level clone detection techniques for comparing statechart actions and activities. We are also interested in devising alternative methods of calculating graph similarity, e.g., by counting the number of identical or similar sub-graphs and more.

References

1. Beuche, D.: Transforming Legacy Systems into Software Product Lines. In: Proc. of SPLC 2011 Tutorial (2011)
2. Borba, P., Teixeira, L., Gheyi, R.: A Theory of Software Product Line Refinement. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 15–43. Springer, Heidelberg (2010)

3. Boucher, Q., Classen, A., Heymans, P., Bourdoux, A., Demonceau, L.: Tag and Prune: a Pragmatic Approach to Software Product Line Implementation. In: Proc. of ASE 2010 (2010)
4. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley (2001)
5. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley (2004)
6. Hosseini, S., Azgomi, M.A.: UML Model Refactoring with Emphasis on Behavior Preservation. In: Proc. of TASE 2008, pp. 125–128 (2008)
7. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU/SEI-90TR-21 (1990)
8. Kastner, C., Apel, S.: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: Proc. of GPCE Wrksp. on Modul., Comp. and Gen. Tech. for PLE (GPLE 2008), pp. 35–40 (2008)
9. Kim, K., Kim, H., Kim, W.: Building Software Product Line from the Legacy Systems: Experience in the Digital Audio and Video Domain. In: Proc. of SPLC 2007 (2007)
10. Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study: Practice Articles. J. of Software Maintenance and Evolution 18(2), 109–132 (2006)
11. Koschke, R., Frenzel, P., Breu, A.P., Angstmann, K.: Extending the Reflection Method for Consolidating Software Variants into Product Lines. Soft. Quality Control 17(4) (2009)
12. Krueger, C.W.: Easing the Transition to Software Mass Customization. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, pp. 282–293. Springer, Heidelberg (2002)
13. Liu, J., Batory, D., Lengauer, C.: Feature Oriented Refactoring of Legacy Applications. In: Proc. of ICSE 2006, pp. 112–121 (2006)
14. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE TSE 30(2), 126–139 (2004)
15. Murphy, G.C., Lai, A., Walker, R.J., Robillard, M.P.: Separating Features in Source Code: an Exploratory Study. In: Proc. of ICSE 2001, pp. 275–284 (2001)
16. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Statecharts Specifications. In: Proc. of ICSE 2007, pp. 54–64 (2007)
17. OMG, `http://www.omg.org/spec/XMI/2.1.1/` (last Accessed: January 2011)
18. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
19. Rubin, J., Chechik, M.: From Products to Product Lines Using Model Matching and Refactoring. In: Proc. of SPLC Wrksp. (MAPLE 2010) (2010)
20. Rubin, J., Chechik, M.: Quality of Behavior-Preserving Product Line Refactorings (2011); Under review
21. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Extraction of Feature Models from Formal Contexts. In: Proc. of SPLC 2011, pp. 4:1–4:8 (2011)
22. Sabetzadeh, M., Easterbrook, S.: View Merging in the Presence of Incompleteness and Inconsistency. Requirement Engineering 11, 174–193 (2006)
23. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse Engineering Feature Models. In: Proc. of ICSE 2011 (2011)
24. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M.: Refactoring UML Models. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 134–148. Springer, Heidelberg (2001)