# Quality of Merge-Refactorings for 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.** In this paper, we consider the problem of refactoring related software products specified in UML into annotative product line representations. Our approach relies on identifying commonalities and variabilities in existing products and further merging those into product line representations which reduce duplications and facilitate reuse. Varying merge strategies can lead to producing several *semantically correct*, yet *syntactically different* refactoring results. Depending on the goal of the refactoring, one result can be preferred to another. We thus propose to capture the goal using a syntactic *quality* function and use that function to guide the merge strategy. We define and implement a *quality*-based *merge-refactoring* framework for UML models containing class and statechart diagrams and report on our experience applying it on three case-studies.

## 1 Introduction

A *software product line (SPL)* is a set of software-intensive products sharing a common, managed set of features that satisfy the specific needs of a particular market segment [4]. SPL engineering practices capitalize on identifying and managing *commonalities* and *variabilities* across the whole product portfolio and promote *systematic software reuse*. SPL commonalities represent artifacts that are part of each product of the product line, while SPL variabilities – those specific to some but not all products. Benefits of applying SPL engineering practices include improved time-to-market and quality, reduced portfolio size, engineering costs and more [4, 9]. Numerous works, e.g., [9], promote the use of SPL practices for model-based development of complex embedded systems. Often, variants of such systems are developed for different customers and are represented and implemented using visual structural and behavioral models.

In reality, however, SPLs often emerge *ad-hoc*, when companies have to release a new product that is similar, yet not identical, to existing ones. Under tight project scheduling constraints, development teams resort to copying artifacts from one of the existing products and later modifying them independently from the original version [15, 18] (the clone-and-own approach).

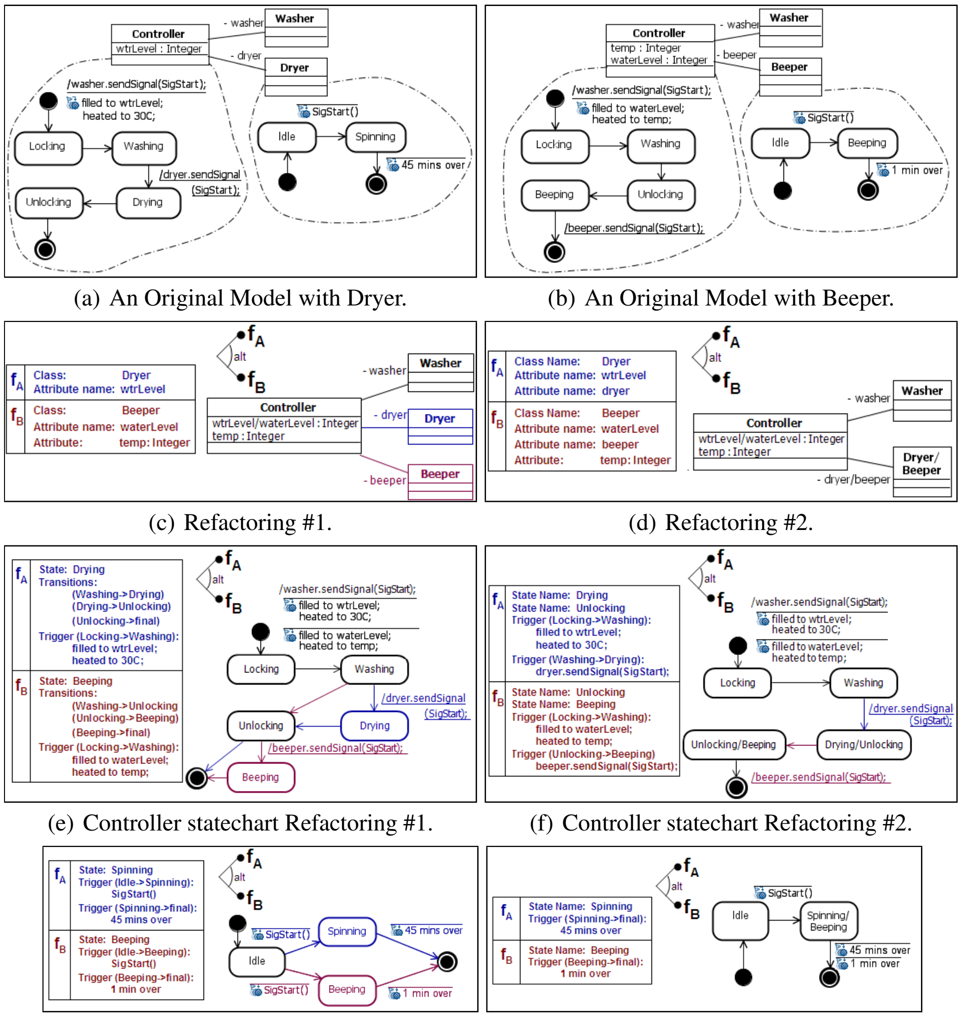
Cloned artifacts require synchronization: changes in one artifact might need to be repeated for all variants. In addition, it is difficult to propagate a new feature implemented in one variant into another or to define a new product by selectively "picking" some, but not all, features from the existing variants. As the result, when product variants are realized via cloning, development and maintenance efforts increase as the number of products grows. To deal with the complexity of SPL development, some approaches, e.g., [2], advocate refactoring legacy cloned products into "single-copy" representations, eliminating duplications and explicating variabilities.

Numerous works provide guidelines and methodologies for building product line representations out of legacy systems, e.g., [14, 8]. Most of such approaches, however, involve a manual review of code, design and documentation of the system aiming at identifying the set of product line features, as well as the set of components which implement these features. This manual step is time-consuming, and, in many cases, impedes adoption of SPL techniques by organizations.

Automated approaches for mining legacy product lines and refactoring them into feature-oriented product line representations have also been proposed [7, 18, 30, 25]. In our earlier work [22, 24], we focused on refactoring model-level cloned product variants and proposed a configurable *merge-refactoring* algorithm, *merge-in*, applicable to refactoring models of different types (e.g., UML, EMF and Matlab/Simulink). Our algorithm identifies similar and different elements of the input models using parameterizable *compare* and *match* operators, and then constructs a refactored model using a *merge* operator. The resulting product line model contains reusable elements representing corresponding merged elements of the original models. In [24], we formally proved that *merge-in* produces *semantically correct* refactorings for *any set* of input models and parameters: a refactored model can derive exactly the set of original products, regardless of particular parameters chosen and implementations of *compare* / *match* / *merge* used, if they satisfy well-defined correctness properties (e.g., *"each element produced by* merge *originates from an element of at least one input model"*).

Varying *merge-in* parameters allows producing different *syntactic* representations of the resulting product line due to different possible ways to match input model elements. All these representations are *semantically* equivalent and derive the same set of products. However, not all possible *syntactic* representations are desirable. Moreover, depending on the goal of the refactoring, one representation might be preferable to another. For example, a goal of the refactoring can be to highlight the variability points between the products, eliminating the "unnecessary" variability and creating a more homogeneous product portfolio. Another can be to maximize the comprehensibility of the resulting model by minimizing variability annotations for elements of a certain type. Yet another can be to reduce the size of the resulting refactoring – this might happen if the models are used for execution or code generation rather than human inspection. These different goals induce different product line representations.

**Example.** Consider the UML model fragments in Fig. 1(a,b) depicting two representative parts of real-life products developed by an industrial partner (since partner-specific details are confidential, we move the problem into a familiar domain of washing machines). Fig. 1(a) shows the `Controller`, `Washer` and `Dryer` classes of a washing machine, together with snippets of `Controller`'s and `Dryer`'s behaviors specified by UML statechart models. The `wtrLevel` attribute of `Controller` is used to specify the desired water level. When the water is filled to that level and heated to $30^oC$, the washing machine controller notifies `Washer` that it can start operating and transitions from the `Locking` to the `Washing` state. After finishing washing, the controller initiates `Dryer` and transitions to the `Drying` state. `Dryer` operates for 45 minutes and returns the control to the `Controller`'s statechart (by sending an appropriate signal which is omitted from the picture to save space). Then, the washing machine is unlocked, and the wash cycle stops. Fig. 1(b) shows a similar washing machine model which lacks the


Controller
wtrLevel : Integer
- washer
Washer
- dryer
Dryer
/washer.sendSignal(SigStart);
filled to wtrLevel;
heated to 30C;
Locking
Washing
Drying
Unlocking
/dryer.sendSignal(SigStart);
SigStart()
Idle
Spinning
45 mins over
temp : Integer
waterLevel : Integer
- beeper
Beeper
filled to waterLevel;
heated to temp;
Beeping
/beeper.sendSignal(SigStart);
1 min over
alt
fA
fB
Class: Dryer
Attribute name: wtrLevel
Class: Beeper
Attribute name: waterLevel
Attribute: temp: Integer
wtrLevel/waterLevel : Integer
Class Name: Dryer
Attribute name: dryer
Class Name: Beeper
Attribute name: beeper
Dryer/Beeper
- dryer/beeper
State: Drying
Transitions: (Washing->Drying) (Drying->Unlocking) (Unlocking->final)
Trigger (Locking->Washing): filled to wtrLevel; heated to 30C;
State: Beeping
Transitions: (Washing->Unlocking) (Unlocking->Beeping) (Beeping->final)
Trigger (Locking->Washing): filled to waterLevel; heated to temp;
State Name: Drying
State Name: Unlocking
Trigger (Washing->Drying): dryer.sendSignal(SigStart);
State Name: Unlocking
State Name: Beeping
Trigger (Unlocking->Beeping) beeper.sendSignal(SigStart);
Unlocking/Beeping
Drying/Unlocking
State: Spinning
Trigger (Idle->Spinning): SigStart()
Trigger (Spinning->final): 45 mins over
State: Beeping
Trigger (Idle->Beeping): SigStart()
Trigger (Beeping->final): 1 min over
State Name: Spinning
Trigger (Spinning->final): 45 mins over
State Name: Beeping
Trigger (Beeping->final): 1 min over
Spinning/Beeping

(a) An Original Model with Dryer. (b) An Original Model with Beeper.

(c) Refactoring #1. (d) Refactoring #2.

(e) Controller statechart Refactoring #1. (f) Controller statechart Refactoring #2.

(g) Dryer and Beeper Refactoring #1. (h) Dryer and Beeper Refactoring #2.

Fig. 1: Fragments of Washing Machine models and some of their refactorings.

dryer but has a beeping function indicating the end of the wash cycle by signalling for 1 minute. In addition, in this model, the `temp` and `waterLevel` attributes control the desired water temperature and water level, respectively.

These two products have a large degree of similarity and can be refactored into annotative SPL representations, where duplications are eliminated and variabilities are explicated. We consider only those refactorings that preserve the behavior of existing products rather than allowing novel feature combinations (e.g., a product with both the dryer and the beeper). Even with this simplification, several choices emerge. For example, the two input models in Fig. 1(a, b) can be combined as shown in Fig. 1(c) where the `Controller` classes of both input models are matched and merged together, while the `Dryer` and the `Beeper` classes are unmatched and thus both copied to the result "as is", together with their corresponding statecharts. Another choice is shown in

Fig. 1(d) where these two classes are matched and merged together, producing either a representation in Fig. 1(g) or in (h). Combining statecharts of `Controller` classes can also result in two possible representations, as shown in Fig. 1(e) and (f). That is, there are six possible refactoring options: Fig 1(c,e), (c,f), (d,e,g), (d,e,h), (d,f,g) and (d,f,h).

In each of the cases, the created models are controlled by a set of features, depicted in the middle upper part of each figure. Since the refactored product line in our example encapsulates only the original input products, we have just two alternative features representing these products – $f_A$ and $f_B$. Product line model elements are *annotated* by these features, as shown in a table on the left-hand side of each figure. The set of annotations specifies elements to be selected given a particular feature selection: selecting $f_A$ filters out all elements annotated with $f_B$, which derives the original input model shown in Fig. 1(a) from each of the refactorings. Likewise, selecting feature $f_B$ derives the original model shown in Fig. 1(b). For a refactoring that aims at maximizing the comprehensibility of the resulting model, the best representation is the one shown in Fig. 1(c, e) since it has the least number of classes and states with variable names and the least number of variable statecharts. However, for a refactoring that aims at reducing the size of the result, the best representation is the one in Fig. 1(d, f, h), as it contains three classes and six states only, compared to the refactoring in Fig. 1(c, e) which has four classes and nine states.

**Contributions.** We consider the problem of integrating several distinct products specified in UML into an annotative product line representation using *merge-refactorings*. **(1)** We argue that there can be multiple syntactically different product line models that represent the same set of products. All such models are *valid*, but not all are *desired*. Explicating the goal of the refactoring can help produce those that better fit the user intention. **(2)** We propose to capture the goal of the refactoring using a quantitative *quality* function, comprised of a set of measurable syntactic metrics. This function is used to evaluate the produced refactorings and to *guide* the *merge-refactorings* process towards a desired result. **(3)** We present an approach for exploring the set of different refactorings with the goal of identifying the one that maximizes the value of a given *quality* function. **(4)** We report on an implementation of a *quality*-based *merge-refactoring* framework for UML models containing class and statechart diagrams, which realizes the *merge-in* algorithm introduced in our earlier work [24]. We use the implemented framework for evaluating the effectiveness of our approach using several example product lines specified in UML, including one contributed by an industrial partner.

The remainder of this paper is organized as follows. The details on annotative product line representations and the *merge-in* refactoring algorithm are given in Sec. 2. Our *quality*-based *merge-refactoring* framework is described in Sec. 3. In Sec. 4, we present an implementation of the framework. We describe our experience applying it to three case studies in Sec. 5. A discussion and a description of related work are given in Sec. 6. Sec. 7 concludes the paper and outlines future research directions.

## 2 Refactoring Software Product Lines

In this section, we describe software product line models annotated by features. We also give the necessary background on *model merging*, used as a foundation of our *merge-in* product line refactoring algorithm, and summarize the *merge-in* algorithm itself.

**Software Product Lines.** SPL approaches can largely be divided into two types: *compositional* which implement product features as distinct fragments and allow generating specific products by composing a set of fragments, and *annotative* which assume that there is one "maximal" representation in which annotations indicate the product features that a particular fragment realizes [11, 3]. A specific product is obtained by removing fragments corresponding to discarded features. Similarly to [3], our experience is that the annotative approach, which reminds code-level `#ifdef` statements, is easier to adopt in practice, as it does not require a paradigm shift in the way software is being commonly developed, especially in the embedded domain. We thus follow this approach here.

A *feature model* is a set of elements that describe product line features and a propositional formula defined over these features to describe relationships between them. A *feature configuration*, defining a product of a product line, is a sub-set of features from the feature model that respect the given relationships. An *annotative product line* is a triple consisting of a feature model, a domain model (e.g., a set of UML classes and statecharts), and a set of relationships that *annotate* elements of the domain model by the features of the feature model. Fig. 1(c-h) present snippets of domain models (right-hand side of each figure) whose elements are connected to features from a feature model (top-middle part of each figure) using annotation relationships (left-hand side of each figure). In this case, features $f_A$ and $f_B$ are alternative to each other, i.e., the propositional formula that 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 the set of elements annotated by features from this configuration. For example, the class diagrams in Fig. 1(a) and Fig. 1(b) can be derived from the product line in Fig. 1(d) under the configurations $\{f_A\}$ and $\{f_B\}$, respectively.

**Model Merging.** Model merging consists of three steps: *compare*, *match* and *merge*.

*Compare* is a heuristic function that calculates the similarity degree, a number between 0 and 1, for each pair of input elements. It receives models $M_1$, $M_2$ and a set of empirically assigned weights which represent the contribution of model sub-elements to the overall similarity of their owning elements. For example, a similarity degree between two classes is calculated as a weighted sum of the similarity degrees of their names, attributes, operations, etc. Comparing `Washer` classes in Fig. 1(a, b) to each other yields 1, as these classes are identical in the presented model fragment. Comparing `Controller` classes yields a lower number, e.g., $0.8$, as the classes have different owned properties and behaviors.

*Match* is a heuristic function that receives pairs of elements together with their similarity degree and returns those pairs of model elements that are considered similar. *Match* uses empirically assigned *similarity thresholds* to decide such similarity. Consider the above example, where `Washing` classes had a calculated similarity degree of $1$ and `Controller` classes had a similarity degree of $0.8$: setting class *similarity threshold* to $0.75$ results in matching both pairs of classes, while setting it to $0.85$ results in matching only the `Washing` classes.

Finally, *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, such that matched elements are unified and present in the result only once. For example, `Controller` classes in Fig. 1(a, b) can be unified as shown on the right-hand side of Fig. 1(e): matched states `Locking`, `Washing` and `Unlocking` are unified, while unmatched states `Drying` and `Beeping` are just copied to the result together with their corresponding transitions. While the *compare* and *match* functions rely on heuristically determined weights and similarity degrees, *merge* is not heuristic: its output is uniquely defined by the input set of matched elements.

***Merging-in* Product Lines.** We now describe the *merge-in* refactoring algorithm [24] that puts together input products into an annotative product-line representation. Constructing an annotative product line model consists of three steps: creating a domain model, creating a feature model, and specifying annotation relationships between the features and the domain model elements. For creation of a domain model, *merge-in* relies on *model merging*, described above. Feature models are created using an approach where features represent the original input products and are defined as alternatives to each other, *so only the original products can be derived from the constructed product line model*. Domain model elements are annotated by these features according to the product(s) that contributed them. For the example in Fig. 1(e), state `Drying` is annotated by feature $f_A$ while state `Beeping` is annotated by $f_B$. State `Washing` is common (it exists in both input models) and thus is annotated by both features. Annotations of common elements are not shown in the figure to save space.

Any input product $M$ can be seen as a "primitive" product line with only one feature $f_M$, one feature configuration $\{f_M\}$, and a set of annotations that relate all model elements to that feature. This representation can derive exactly one product – $M$. Thus, the most generic form of the *merge-in* operator obtains as input two (already constructed) product lines, each of which can be a "primitive" product line representing one input model. For example, when combining the two products in Fig. 1(a, b), we implicitly convert each of them into a product line and then *merge-in* them together. One possible outcome of that process is shown in Fig. 1(c, e), where the features representing the original models are denoted by $f_A$ and $f_B$ and defined as alternatives. In this case, `Dryer` and `Beeper` classes are unmatched.

Varying *compare* and *match* parameters, as well as varying the order in which input models are combined, defines the exact shape of the refactoring outcome. Two products in Fig. 1(a, b) can also be combined as shown in Fig. 1(d, f), where a lower class *similarity threshold* results in `Dryer` and `Beeper` classes being matched and merged.

All possible refactorings constructed by the algorithm are semantically "correct", each deriving the exact set of input models, regardless of the parameters chosen and regardless of the order in which input products are *merged-in*. The correctness of the *merge-in* operator relies on "reasonable" behavior of model *compare*, *match* and *merge* algorithms. Formal correctness properties of those algorithms are specified in [24].

## 3 Quality-Based Merge-Refactoring Framework

Even though all refactorings produced by the *merge-in* algorithm are *semantically equivalent and correct*, not all refactorings are desirable: depending on the goal of the refactoring, one representation can be preferred to another. The main objectives of our *quality*-based product line *merge-refactoring* framework are thus to (1) allow the user to explicate the goal of the refactoring process and (2) drive the refactoring process towards the result that best fits the user intention. We depict our approach in Fig. 2 and describe it below.

## 3.1 Explicating The Refactoring Goal

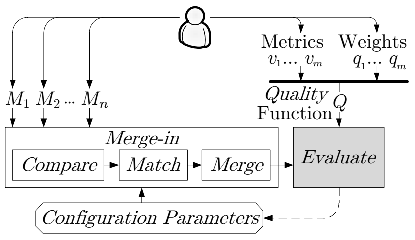


Fig. 2: Merge-Refactoring Framework.

We explicitly capture the goal of the refactoring using a quantitative *quality* function. Since in our work we focus on *syntactic* properties of merge-refactorings, the *quality* function is built from a set of *metrics* – syntactically measurable indicators representing specific refactoring objectives (see Fig. 2). Typically, such metrics can assess the size of the resulting model, determine the degree of object coupling, cohesion in methods, weighted number of methods per class and more [17]. The metrics can be reused across different organizations and domains. Each concrete *quality* function $Q$ assigns *weights* $q_1 \dots q_m$ to different metrics $v_1 \dots v_m$, indicating their "importance" in the context of a specific application domain and allows the user to specify the desired outcome of the refactoring in an explicit manner.

More formally, given a product line model $\mathcal{PL}$, the *quality* function returns a number between 0 and 1, representing $\mathcal{PL}$'s "quality":

$$\textit{quality}(\mathcal{PL}, \mathbb{V}, \mathbb{Q}) = \sum_{i=1 \dots n} q_i * v_i(\mathcal{PL}),$$

where $\mathbb{V} = v_1, \dots, v_n$ is a set of measurable metrics that, given $\mathcal{PL}$, produce a number between 0 and 1, and $\mathbb{Q} = q_1, \dots, q_n$ is a set of metrics' weights.

**Examples of Possible Quality Functions.** We discuss two specific *quality* functions. The goal of the first one, $Q_1$, is to minimize the size of the resulting model. Since we assume that there is a large degree of similarity between input models that represent *related* products of a product line, aiming to reduce the total number of elements in the result leads to a reduction of duplications which, in turn, helps avoid repeating modifications for all variants.

We define our notion of model size using the number of classes, attributes, states and transitions. Specifically, the metrics $v_1$-$v_4$ listed in the first four rows of Table 1 measure the overall reduction of the size of the produced model when compared to the inputs. To construct $Q_1$, we assign these metrics equal weights, considering them equally important, as specified in the second to last column of Table 1. $Q_1$ prefers models that are as compact as possible, e.g., the refactoring in Fig. 1(d, f, h).

Our second goal is to produce refactorings that are the easiest for the user to comprehend. The work of [6, 5] makes an argument that an increase in the size of UML models (specifically, the number of classes, aggregations, states and transitions) leads to an increase of cognitive complexity. The authors validate this claim using controlled experiments involving human subjects. However, neither these experiments nor our early work [22] considered *variability* aspects of the annotative product line representations. For example, while they minimize the size of the model, both possible merges of the `Dryer` and the `Beeper` classes in Figures 1(g) and (h) contain $66\%$ and $50\%$ variable

Table 1: Quality metrics.

| Metric | | Objective | Weight | |
|---|---|---|---|---|
| | | | $Q_1$ | $Q_2$ |
| $v_1$ | % of reduction in # of classes | Reduce the number of classes. (out of the total number of classes in input.) | 0.25 | 0.125 |
| $v_2$ | % of reduction in # of attributes | Reduce the number of class attributes. (out of the total number of attributes in input.) | 0.25 | 0.125 |
| $v_3$ | % of reduction in # of states | Reduce the number of states. (out of the total number of states in input.) | 0.25 | 0.125 |
| $v_4$ | % of reduction in # of transitions | Reduce the number of transitions. (out of the total number of transitions in input.) | 0.25 | 0.125 |
| $v_5$ | % of common attributes | Reduce the percentage of variable class attributes (out of the total number of the class attributes.) | 0.0 | 0.17 |
| $v_6$ | % of common states | Reduce an average percentage of variable states (out of the total number of states in a statechart.) | 0.0 | 0.17 |
| $v_7$ | % of common transitions | Reduce an average percentage of variable transitions (out of the total number of transitions in a statechart.) | 0.0 | 0.16 |

states (i.e., states annotated by features), respectively. The merge of the `Controller` classes in Fig. 1(f) contains $50\%$ variable states as well.

We believe that the higher is the number of common elements in a merged model, the easier it is to understand. We thus define a second *quality* function, $Q_2$, to combine size minimization with encouraging those refactorings which result in models with a high degree of commonalities: classes with a significant number of common attributes and statecharts with a significant number of common states and transitions. The metrics $v_5$-$v_7$ of Table 1 are designed for that purpose. They are calculated by counting the percentage of *common* sub-elements for a certain element in the model, i.e., those sub-elements that are annotated by all product line features. To achieve a reasonable degree of merging while discouraging too much variability, $Q_2$ gives the combination of four size-based metrics $v_1$-$v_4$ and the combination of three variability-based metrics $v_5$-$v_7$ equal importance (see the last column of Table 1). This *quality* function prefers the refactoring in Fig. 1 (c, e).

We use both $Q_1$ and $Q_2$ to evaluate refactorings of our case-study models in Sec. 5.

## 3.2 Constructing the desired refactorings

Since a *quality* function captures the criteria that are to be used when performing the *merge-refactoring* process, it could also be used to guide the process towards the desired result. As stated in Sec. 2, refactorings produced by the *merge-in* algorithm differ by the way input model elements are matched and merged, which is controlled by the *merge-in configuration parameters*. Modifying these parameters, e.g., increasing weight of state name similarities during *compare*, can result in the refactoring shown in Fig. 1(e). Instead, if we give more weight to the structural similarity of states, i.e., their distance to the initial and the final states and, recursively, the similarity of their neighbors [19], we get the result in Fig. 1(f). Likewise, lowering the class *similarity threshold* can result in a refactoring where the `Dryer` and the `Beeper` classes are matched and merged together, in addition to merging the `Controller` classes, as shown in Fig. 1(d).

Obviously, *merge-in* parameters cannot be decided universally because their values depend on the nature of the refactored product line and the objective of the *quality* function. It is also unreasonable to assume that the user can set and adjust these parameters manually. Moreover, generating *all* possible refactorings and evaluating them based on the given *quality* function is not a feasible approach as it does not scale well.

We thus need a systematic way for identifying those values of *merge-in* parameters that result in an optimal refactoring w.r.t. the given *quality* function $Q$. In our work, we propose doing so by treating parameter selection as a classical optimization problem [21], using the chosen *quality* function as an *objective function* for an optimization technique. The process (1) uses an optimization heuristic to set values of *merge-in* parameters, (2) produces the corresponding refactoring, subject to these parameters, (3) evaluates it using $Q$, and repeats until a result of the desired quality is reached (or a certain fixed number of iterations is performed). That is, different refactorings are generated by the *merge-in* algorithm based on the values of *compare weights* and *similarity thresholds* that are set using an optimization algorithm aimed to maximize the value of $Q$. Only the resulting "optimal" refactoring is returned to the user. The overall efficiency of the approach is as good as the chosen optimization algorithm because the latter selects the values of parameters for the next iteration.

## 4 Implementation

In this section, we describe our implementation of the *merge-in* algorithm, used as a foundation of the *merge-refactoring* framework, as well as our approach for setting the *merge-in* parameters. We focus our work on systems represented and implemented with UML models containing class and statechart diagrams – a common choice in automotive, aerospace & defense, and consumer electronics domains, where such models are often used for full behavioral code generation (e.g., using IBM Rhapsody[1]).

The core part of the *merge-in* algorithm is the *compare* function which receives two UML elements of the same type and returns their similarity degree – a number between 0 and 1. To implement *compare*, we started with existing comparison algorithms for UML classes [29, 13] and statecharts [19]. These algorithms calculate the similarity degree recursively, using formulas that assign empirically defined weights to similarity degrees of appropriately chosen sub-elements.

None of the existing algorithms combined information obtained by analyzing both structural and behavior models together: comparing classes did not take into account information about similarity of their corresponding statecharts. We thus extended class comparison by considering behavior information, obtained by comparing statecharts to each other, and combining it with structural information by giving them equal weights. We also extended the statechart comparison algorithm proposed in [19] to consider state `entry` and `exit` actions, state `do` activities and actions on transitions, as those were used in the real-life model provided by the industrial partner.

Based on elements' similarity degrees generated by *compare*, our implementation of *match* "greedily" selects similar elements that are above a given threshold. *Merge* further combines elements deemed similar while explicating variabilities. We use the *union-merge* [26] approach to implement the *merge* function. It unifies matched elements and copies unmatched elements "as is" to the result. Our *merge* implementation is an adaptation of static union merge of `TReMer+`[2], extended to deal with annotations

[1] http://www-01.ibm.com/software/awdtools/rhapsody/

[2] http://se.cs.toronto.edu/index.php/TReMer+

of domain model elements by features, as discussed in Sec. 2. We use IBM Rational Software Architect[3] (RSA) as our modeling environment, allowing us to reuse existing Java-based algorithms. Rhapsody models supplied by our industrial partner were first exported into UML 2.0 XMI format and then imported into RSA.

For adjusting *merge-in* parameters, we have implemented a version of the local search optimization technique [21] where the space of possible refactorings is explored by changing one parameter value at a time (hill-climbing). After evaluating the resulting refactoring and adjusting this value, we move to the next one, until all values are set. While this algorithm can miss the best result (global maximum) because it does not revisit the decisions that were already made, it is shown to be quite effective in practice for finding a "good" result (local maximum) in an efficient manner. We demonstrate the effect of adjusting the class *similarity threshold* in Sec. 5.

We *merge-in* the most similar models first: similarity degrees of all inputs – models of individual products or already constructed intermediate product lines – are evaluated, and those with the highest similarity degrees are combined first. Intuitively, merging more similar models first helps decrease the size and the number of variable elements in the result.

During our experiments, we noted that different values of *compare weights* and *similarity thresholds* can produce the same refactoring and thus the same *quality* measurements. Since our goal is to maximize a given *quality* function, any of the assignments that produce the desired result is appropriate.

## 5 Experience

In this section, we report on our experience applying the *quality*-based *merge-refactorings*. Our goal is to validate the feasibility of the approach for UML models in the embedded domain. In particular, we are interested in demonstrating the applicability and effectiveness of the proposed methodology for adjusting *merge-in* parameters for realistic models containing UML class and statechart diagrams, based on a given *quality* function. In what follows, we describe our subject product lines and present our results.

**Subjects.** We applied our refactoring approach to three sets of related products. The first is the Washing Machine example, built by us to mimic a partner's model and to highlight its characteristics. A snippet of this example is presented in Fig. 1 and the full version is available in [23]. The Washing Machine product line contains three different products, with a high degree of overlap in the set of classes comprising them. Specifically, each product has six classes, out of which three are identical across products (`Motor`, `Faucet` and `Detergent Supplier`), two are similar to each other in all three products (`Controller` and `Washer`), and one class in each product carries a unique product-specific functionality (either `Dryer`, `Beeper` or `Timer`). Also, statecharts of similar classes have similar structures.

The second example, Microwave Oven, has been introduced by Gomaa in [9]. It includes four different, although very similar, variants of the timer control class and their corresponding statecharts.

The final example comes from the Consumer Electronics (CE) space, contributed by an industrial partner. Here, we focus on seven behavior-intensive product components

[3] http://www-01.ibm.com/software/awdtools/swarchitect/

Table 2: Varying Class Similarity Threshold $S_{\text{Class}}$.

| | | Washing Machine | | | | | | | Microwave Oven | | | | CE Product Line | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| metrics | | orig. | 0.7 | 0.75 | 0.78 | 0.8 | 0.85 | 0.9 | orig. | 0.7 | 0.75-0.85 | 0.9 | orig. | 0.6 | 0.65 | 0.7 | 0.75 | 0.8 |
| $v_1$ | #classes | 18 | 6 | 7 | 8 | 9 | 11 | 12 | 8 | 2 | 3 | 5 | 45 | 14 | 15 | 17 | 20 | 27 |
| $v_2$ | #attributes | 25 | 10 | 12 | 14 | 20 | 25 | 25 | 4 | 1 | 2 | 4 | 104 | 56 | 56 | 75 | 84 | 80 |
| $v_5$ | #var.attributes | - | 3 | 4 | 5 | 6 | 4 | 0 | - | 0 | 2 | 4 | - | 43 | 43 | 26 | 32 | 8 |
| $v_3$ | #states | 43 | 18 | 20 | 22 | 28 | 38 | 43 | 18 | 7 | 9 | 18 | 448 | 177 | 151 | 211 | 374 | 412 |
| $v_6$ | #var.states | - | 6 | 6 | 5 | 4 | 0 | 0 | - | 1 | 0 | 0 | - | 56 | 64 | 31 | 13 | 4 |
| $v_4$ | #transitions | 56 | 28 | 31 | 34 | 40 | 51 | 56 | 44 | 16 | 24 | 44 | 944 | 245 | 260 | 402 | 573 | 640 |
| $v_7$ | #var.transitions | - | 19 | 19 | 18 | 12 | 0 | 0 | - | 2 | 4 | 0 | - | 77 | 85 | 31 | 19 | 8 |
| Q1 | | - | 0.587 | 0.528 | 0.469 | 0.333 | 0.148 | 0.083 | - | 0.686 | 0.505 | 0.065 | - | 0.646 | 0.635 | 0.496 | 0.351 | 0.284 |
| Q2 | | - | 0.565 | 0.560 | 0.572 | 0.533 | 0.523 | 0.541 | - | 0.797 | 0.561 | 0.372 | - | 0.640 | 0.650 | 0.678 | 0.601 | 0.623 |

which together contain 45 classes, 104 attributes, 448 states and 944 transitions. The number of classes implementing each component ranges between 2 and 14. The number of statecharts in each component ranges between 1 and 3, with the number of states and transitions for a statechart ranging between 20 and 66 states, and 31 and 81 transitions, respectively. Of the seven components, three have a similar structure and a similar set of elements; another pair of components also contains elements that are similar to each other (but less similar to the components of the first cluster), and the remaining two components are not similar to the rest.

Space limitations and verbosity of UML models do not allow us to include pictorial illustrations of the examples. Thus, we limit the presentation to the statistical data about the case studies. The complete models for the first two examples are available in [23]. Since we cannot share details of the CE model, we built our first example, the Washing Machine, to be similar.

**Results.** To evaluate the effectiveness of our *quality*-based merge-refactoring approach, we analyzed different refactorings produced by varying *compare weights* and *similarity thresholds*, and evaluated them using *quality* functions $Q_1$ and $Q_2$ introduced in Sec. 3. As a starting point, we used empirically determined weights specified in [29, 13, 19]. We updated the weights to combine structural and behavior information when comparing classes and to take into account additional statechart elements, as described in Sec. 4. For the *similarity thresholds*, we started with the assumption that elements with the similarity degree lower than $0.5$ are significantly different and should not be combined. For statecharts, we refined these estimates using the thresholds empirically determined in [19]. The values of weights and thresholds that we used are summarized in [23].

For illustration purposes, in this section we vary the class *similarity threshold* between $0.4$ and $1$, iteratively incrementing its value by $0.01$, and evaluate the produced results using our *quality* functions. Table 2 presents the total number of elements as well as the number of *variable* elements of each type in the resulting refactoring. To save space, we show only *distinct* refactorings, omitting those that are equivalent to the presented ones. For example, in the Washing Machine case, all refactorings produced with class *similarity thresholds* between 0.4 and 0.7 are identical, and we only show the latest. In addition, the *orig* column reports the total number of input elements for each of the case studies. It is used to compare the result of the refactoring to the original models and to normalize the collected metrics during *quality* computation. A full description of the refactored product line models that were produced for each step of the first two case-studies is available in [23].

The results demonstrate that increasing the value of the class *similarity threshold* results in decreasing the value of $Q_1$ in all case studies because this function prefers refactorings that are as compact as possible: as the class *similarity threshold* increases, fewer classes are matched and merged, and the number of elements in the result grows. $Q_2$, however, does not exemplify such linear behavior because it balances the reduction in size with the goal of merging only those elements that are indeed similar. For example, when refactoring the Washing Machine, the result preferred by $Q_1$ is obtained by setting the class *similarity threshold* to $0.7$ or lower, which causes merging of as many classes as possible, including those that are dissimilar (e.g., the one in Fig. 1(d)). This produces state machines with a large percentage of variable states and transitions. $Q_2$ prefers the solution produced when the *similarity threshold* is set to $0.78$, which merges only elements with a high degree of commonality (e.g., see Fig. 1(c)). When the class *similarity threshold* is high ($0.9$), only identical classes got merged. A large number of classes, states and transition in the resulting model is captured by a low calculated value for both $Q_1$ and $Q_2$, since both of them are designed to minimize the size of the result.

For the Microwave Oven example, both $Q_1$ and $Q_2$ prefer the solution found when the class *similarity threshold* is set to 0.7 or lower (see Table 2). Since all four variants of the timer control class in this example are very similar, these classes are all merged together in the resulting refactoring. The percentage of variable states and transitions in this solution remains small, and the overall reduction in their total number is significant.

Recall that our third example had two clusters of similar components (and two other components, different from the rest). The refactoring that identifies and merges components in these clusters is produced when the class *similarity threshold* is set to $0.7$. This refactoring also maximizes the value of $Q_2$. Similarly to the Washing Machine case, lower threshold values produce more merges resulting in a high number of variable attributes, states and transitions (and thus, lower values of $Q_2$), while higher thresholds result in a large number of elements in the resulting model (and thus, lower values of both $Q_1$ and $Q_2$).

In summary, we found that in all of the above cases, *quality* functions were able to distinguish between different refactorings as desired and thus were appropriate to help "drive" the refactoring process towards the preferable result. Our third case study also showed that differences in the computed *quality* values became more pronounced as models got bigger. Furthermore, the refactorings that were produced in our examples under the strategy that maximizes the value of $Q_2$ were identical to those constructed manually by a domain expert. This encouraging result makes us believe that our *quality*-based *merge-refactorings* approach is effective for the creation of annotative product line representations from a set of existing systems.

**Threats to Validity.** Threats to external validity are most significant for our work. These arise when the observed results cannot generalize to other case studies. Because we used a limited number of subjects and *quality* functions, our results might not generalize without an appropriate tuning. However, we attempted to mitigate this threat by using a real-life case study of considerable size as one of our examples. Thus, even though preliminary, our results show that the approach, perhaps with some additional tuning, is effective for finding good refactorings of large-scale systems.

In addition, we limit the scope of our work to physical systems in the embedded domain, where number of product variants usually does not exceed tens. The approach might not scale well to other domains, where hundreds of product variants are possible.

However, we believe that the scalability issue mostly relates to the annotative product line representation itself, rather than to our attempt to distinguish between different representations.

## 6 Discussion and Related Work

**Product Line Refactoring Approaches**. Several existing approaches aim at building product lines out of legacy artifacts, e.g., [8]. These approaches mainly provide guidelines and methodologies for identifying features and their related implementation components rather than build tool-supported analysis mechanisms. Some works also report on successful experience in manual re-engineering of legacy systems into feature-oriented product lines, e.g., [14].

Koschke et al. [15] present an automated technique for comparing software variants at the architectural level and reconstructing the system's static architectural view which describes system components, interfaces and dependencies, as well their grouping into subsystems. Ryssel et al. [25] introduce an automatic approach to re-organize Matlab model variants into annotative representations while identifying variation points and their dependencies. Yoshimura et al. [30] detect variability in a software product line from its change history. None of the above approaches, however, takes into account *quality* attributes of the constructed results nor attempt to distinguish between the different refactorings based on the refactoring goal.

**Product Line Quality**. Oliveira et al. [20] propose a metric suite to support evaluation of product line architectures based on McCabe's *cyclomatic complexity* of their core components, which is computed using the control flow graph of the program and measures the number of linearly independent paths through a program's source code. Her et al. [10] suggest a metric to measure *reusability* of product line core assets based on their ability to provide functionality to many products of the same SPL, the number of SPL variation points that are realized by an asset, the number of replaceable components in a core asset and more. Hoek et al. [28] describe metrics for measuring *service utilization* of SPL components based on the percentage of provided and required services of a component. While these works allow measuring reusability, extensibility and implementation-level complexity of product line core assets, they do not discuss the structural complexity of annotative SPL models nor allow comparing different annotative product line models and distinguishing between them based on their representation properties. Trendowicz and Punter [27] investigate to which extend existing quality modeling approaches facilitate high quality product lines and define requirements for an appropriate quality model. They propose a goal-oriented method for modeling quality during the SPL development lifecycle, but do not propose any concrete metrics.

Numerous works, e.g., [6, 5], propose software metrics for evaluating quality of UML models. While we base our approach on some of these works, they are not designed for UML models that represent software product lines and do not take variability aspects into account.

Finally, some works discuss characteristics of feature implementations in code, such as feature cohesion and coupling [1] or granularity, frequency and structure of preprocessor annotations [12, 16]. However, these works are not easily generalizable to address the issue of structural complexity of models.

## 7 Conclusions and Future Work

Understanding and refactoring existing legacy systems can promote product line adoption by industrial organizations which have made a significant investment in building and maintaining these systems, and are not ready to abandon them for "starting from scratch". Since these systems are usually very large, automation becomes a necessity.

In this paper, we focused on integrating distinct products specified in UML into an annotative product line representation. We argued that multiple *syntactically different* yet *semantically equivalent* representations of the same product line model are possible, and *the goal of the refactoring induces which one is preferable*. We suggested an approach for guiding the refactoring process towards a result that fits best the user's intention, as captured by a syntactic *quality* function. We implemented a refactoring algorithm based on *model merging* and used it as the foundation of our *merge-refactoring* framework. We evaluated the proposed *quality*-based *merge-refactoring* approach on a set of case-studies, including a large-scale example contributed by an industrial partner. We believe that our work promotes automation of product line refactoring and reasoning about refactoring alternatives.

For future work, we are interested in enhancing our understanding of product line *quality* considerations which can help with assessing different product line model representations, produced either automatically or manually. The *quality* functions can be extended to consider additional quality attributes, allow the user to set and/or interactively choose different quality goals for different regions within the model, incorporate user feedback and more. Performing user studies for evaluating quality of annotative product line models can also be a subject of future work.

In addition, we are interested in exploring more sophisticated refactoring techniques that are able to detect fine-grained features in the combined products. This will allow creating new products in the product line by "mixing" features from different original products, e.g., the dryer and the beeper features from the models in Fig. 1. We also plan to further improve our *match* algorithms by allowing the user to affect results of this function, e.g., by setting negative matches. Exploring the use of more advanced optimization techniques, such as cross-entropy for adjusting *compare* and *match* parameters is also a subject for possible future work.

**Acknowledgments.** We thank Andrei Kirshin (IBM Research) for useful discussions and the anonymous reviewers for their insightful comments.

## References

1. S. Apel and D. Beyer. Feature Cohesion in Software Product Lines: an Exploratory Study. In *Proc. of ICSE'11*, pages 421–430, 2011.
2. D. Beuche. Transforming Legacy Systems into Software Product Lines. SPLC'11 Tutorial.
3. Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and Prune: a Pragmatic Approach to Software Product Line Implementation. In *Proc. of ASE'10*, 2010.
4. P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. 2001.
5. J. A. Cruz-Lemus, M. Genero, and M. Piattini. Using Controlled Experiments for Validating UML Statechart Diagrams Measures. In *Proc. of Wrksp. on Soft. Proc. and Prod. Measure.*, volume 4895 of *LNCS*, pages 129–138, 2008.

6. M. Esperanza Manso, J. A. Cruz-Lemus, M. Genero, and M. Piattini. Empirical Validation of Measures for UML Class Diagrams: A Meta-Analysis Study. In *Proc. of MODELS'08 Workshops*, volume 5421 of *LNCS*, pages 303–313, 2009.
7. D. Faust and C. Verhoef. Software Product Line Migration and Deployment. *Journal of Software Practice and Experiences*, 30(10):933–955, 2003.
8. S. Ferber, J. Haag, and J. Savolainen. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *Proc. of SPLC'02*, pages 235–256, 2002.
9. H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley, 2004.
10. J. S. Her, J. H. Kim, S. H. Oh, S. Y. Rhew, and S. D. Kim. A Framework for Evaluating Reusability of Core Asset in Product Line Engineering. *Information and Software Technology*, 49(7):740 – 760, 2007.
11. C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of GPCE Workshops (McGPLE'08)*, pages 35–40, 2008.
12. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. of ICSE'08*, pages 311–320, 2008.
13. U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. In *Software Engineering*, volume 64 of *LNI*, pages 105–116, 2005.
14. R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. 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.
15. R. Koschke, P. Frenzel, A. P. Breu, and K. Angstmann. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *Software Quality Control*, 17(4), 2009.
16. J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. of ICSE'10*, 2010.
17. J. A. McQuillan and J. F. Power. On the Application of Software Metrics to UML Models. In *Proc. of MODELS'06 Workshops*, pages 217–226, 2006.
18. T. Mende, R. Koschke, and F. Beckwermert. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *J. of Software Maintenance and Evolution*, 21(2), 2009.
19. S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *Proc. of ICSE'07*, pages 54–64, 2007.
20. E. Oliveira Junior, J. Maldonado, and I. Gimenes. Empirical Validation of Complexity and Extensibility Metrics for Software Product Line Architectures. In *Proc. of SBCARS'10*, 2010.
21. P. Pardalos and M. Resende. *Handbook of Applied Optimization*. 2002.
22. J. Rubin and M. Chechik. From Products to Product Lines Using Model Matching and Refactoring. In *Proc. of SPLC Wrksp. (MAPLE'10)*, 2010.
23. J. Rubin and M. Chechik. http://www.cs.toronto.edu/˜mjulia/PLRefactoring/, 2012.
24. J. Rubin and M. Chechik. Combining Related Products into Product Lines. In *Proc. of FASE'12*, pages 285–300, 2012.
25. U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic Variation-Point Identification in Function-Block-Based Models. In *Proc. of GPCE'10*, pages 23–32, 2010.
26. M. Sabetzadeh and S. Easterbrook. View Merging in the Presence of Incompleteness and Inconsistency. *Requirement Engineering*, 11:174–193, June 2006.
27. A. Trendowicz and T. Punter. Quality Modeling for Software Product Lines. In *Proc. of ECOOP Workshops (QAOOSE'03)*, 2003.
28. A. van der Hoek, E. Dincel, and N. Medvidovic. Using Service Utilization Metrics to Assess the Structure of Product Line Architectures. In *Proc. METRICS'03*, pages 298–308, 2003.
29. Z. Xing and E. Stroulia. UMLDiff: an Algorithm for Object-Oriented Design Differencing. In *Proc. of ASE'05*, pages 54–65, 2005.
30. K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. FAVE: Factor Analysis Based Approach for Detecting Product Line Variability from Change History. In *Proc. of MSR'08*, pages 11–18, 2008.