GuyDance: Guiding Configuration Updates for Product-Line Evolution

Michael Nieke
TU Braunschweig
Braunschweig, Germany
m.nieke@tu-bs.de

Gabriela Sampaio
Imperial College London
London, United Kingdom
g.sampaio17@imperial.ac.uk

Thomas Thüm
Ulm University
Ulm, Germany
thomas.thuem@uni-ulm.de

Christoph Seidl
ITU Copenhagen
Copenhagen, Denmark
chse@itu.dk

Leopoldo Teixeira
Federal University of Pernambuco
Recife, Brazil
lmt@cin.ufpe.br

Ina Schaefer
TU Braunschweig
Braunschweig, Germany
i.schaefer@tu-bs.de

ABSTRACT

A product line is an approach for systematically managing configuration options of customizable systems, usually by means of features. Products are generated by utilizing configurations consisting of selected features. Product-line evolution can lead to unintended changes to product behavior. We illustrate that updating configurations after product-line evolution requires decisions of both, domain engineers responsible for product-line evolution as well as application engineers responsible for configurations. The challenge is that domain and application engineers might not be able to talk to each other. We propose a formal foundation and a methodology that enables domain engineers to guide application engineers through configuration evolution by sharing knowledge on product-line evolution and by defining configuration update operations. As an effect, we enable knowledge transfer between those engineers without the need to talk to each other. We evaluate our method by providing formal proofs that show product behavior of configurations can be preserved for typical evolution scenarios.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Software evolution**; **Maintaining software**; Software configuration management and version control systems.

KEYWORDS

software product line, configuration, evolution

ACM Reference Format:
Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. 2020. GuyDance: Guiding Configuration Updates for Product-Line Evolution. In *24th ACM International Systems and Software Product Line Conference Companion (SPLC '20 Companion), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3382026.3425769

1 INTRODUCTION

Configurable software allows customization of software products to fit users' requirements. For instance, cars and their software can be configured by customers through a web configurator of the car manufacturer [29] and the Linux kernel can be custom-tailored by selecting from more than 21,000 configuration options [18]. A product line is a concept for managing configurable software systems and their configuration options in terms of features [2, 10, 19, 23]. A *configuration* of a product line is a set of selected features. The set of available features and their valid combinations are often captured using a *feature model* [10]. A *mapping* uses Boolean formulas to associate features with reusable artifacts or parts thereof (e.g., through preprocessor statements in C++ code and a configuration sets variables for compile time variability). Using these artifacts, a *product* can be generated automatically for a given configuration [2, 7]. In the product-line life cycle, two main roles are involved: during domain engineering, *domain engineers* specify feature models and mappings [19]; during application engineering, *application engineers* define configurations to generate products.

In the process of product-line evolution, domain engineers may change the set of features, artifacts, and the mapping [11]. This can lead to unintended changes to product behavior [9]. For instance, if a feature `A` is merged into another feature `B`, configurations selecting only `B` and not `A` represent different product behavior before and after evolution. Previous research identified the need of practitioners to know how changes impact existing configurations and that it is pivotal to know whether a system operates as expected after evolution [3, 12]. Thus, configuration evolution must be in line with product-line evolution. Application engineers are left with the task of detecting and fixing problems manually with existing configurations used in the field, which is time consuming and error prone [31].

When trying to update configurations to new product-line versions, domain engineers and application engineers face problems in sharing their knowledge with each other: first, with long product-release cycles, the time span between evolution of product lines and configurations can exceed months or years so that detailed knowledge of the evolution may be lost; second, domain and application engineers may not know each other, which creates a communication barrier [5]. For instance, a domain engineer developing the Linux kernel does not know all end-users configuring it. Hence, domain engineers do not necessarily know which configurations

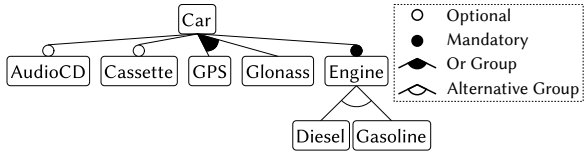


Figure 1: Feature model of the running example.

are actually used and may not be aware of the requirements the generated product has to fulfill. Similarly, application engineers do not know why and how product-line evolution was performed and, in isolation, cannot decide on how to change their configurations, especially if multiple evolution steps have been performed. Leaving application engineers with the task of updating their configurations is bad practice and often leads to misconfiguration [33, 34]. Previous research attempts provide automated fixes for configurations which may inadvertently alter product behavior [30–32, 34]. Even if applicable, these approaches assume that engineers in isolation are able to choose a suitable fix. However, depending on the evolution neither domain engineers nor application engineers are able to adapt configurations without each other.

Apart from lack of communication mechanisms, the number of application engineers typically is significantly higher than domain engineers, which leads to a high communication overhead or makes it even impossible for domain engineers to talk to each application engineer [5]. For instance, thousands of application engineers configure the Linux kernel but only a few develop it. Moreover, industry reports that, for some systems, configuration logic changes almost weekly [3]. One of our industry partners reports even 200 changes in a year. Without automating the communication between domain and application engineers, updating configurations requires massive communication efforts which quickly become infeasible.

In this paper, we present *guided configuration evolution*, providing guidance for domain and application engineers when updating configurations to a new product-line version. Our main goal is to enable knowledge transfer from domain engineers to application engineers without the need to talk to each other. To this end, domain engineers define guidance in the form of instructions for application engineers on how to update configurations to best cope with product-line evolution – ideally maintaining product behavior fully automatically. We propose a formal foundation and a general methodology allowing domain engineers to define guidance for application engineers to update their configurations. Such guidance consists of a rationale for product-line evolution and concrete update suggestions for configurations that can be applied automatically. Optimally, product behavior is preserved after evolution but, even if this cannot be achieved, application engineers are made aware and can make an informed decision on how to adapt configurations. Guidance is defined once by a domain engineer and can be reused by an arbitrary number of application engineers. In addition, domain engineers do not have to define guidance for each individual configuration but for large sets of configurations. To enable reuse, our methodology allows to define templates for guidance of typical evolution scenarios independent of their specific application context. We illustrate the use of the methodology by means of three exemplary pre-defined evolution templates which define how the set of features and the mapping evolve. In our evaluation, we formally prove that we are able to preserve product behavior of configurations for typical evolution scenarios using our methodology. In summary, we make the following contributions:

- We propose a formal foundation for domain engineers to express evolutionary changes to configurations.
- We define a methodology with a prototypical tool *GuyDance*[1] enabling domain engineers to guide application engineers in updating configurations.
- We provide three example evolution templates to support domain engineers, which illustrate the methodology and the formalism.
- We formally prove soundness of the templates by establishing behavior preservation for subsets of configurations.[2]

2 BEHAVIOR PRESERVATION

Given a configuration, the product generated before and after product-line evolution may behave differently due to changes to artifacts that are mapped to features. In the following, we define our notion of product behavior. To this end, we first introduce basic product-line concepts by means of a running example of a feature model for a car product line depicted by Figure 1. We adapt existing notions and formalisms for product lines [4, 21]. We formalize a feature model F as the set of all features. We abstract from feature relations or other constraints, as our notion of product behavior is independent of such relations. A configuration C is a set of selected features such that $C \subseteq F$. Each feature $f \notin C$ is implicitly deselected.

To generate a product for a given configuration, it is necessary to know which reusable artifacts have to be selected. The set I contains all reusable artifacts. For instance, the `Engine` feature can be realized using a plug-in `car.engine`. In a *mapping*, features are related to reusable artifacts [10]. For instance, for the running example, a mapping with preprocessor directives could look like: `#if Engine <code> #endif`. We abstract from concrete implementation and mapping techniques. We consider a mapping $M : ac \rightarrow \mathcal{P}(I)$ as a function relating features in terms of an application condition ac, being a Boolean formula over features, and a set of mapped artifacts. For instance, a mapping entry could look like: $M(\texttt{GPS} \vee \texttt{Glonass}) = \{\texttt{car.positioning}\}$.

Finally, we consider a product line as a triple $PL = (F, I, M)$ with the feature model F, the set of reusable artifacts I, and the mapping M. A product can be generated by composing all reusable artifacts that are collected using the mapping and a configuration. We define a product of a configuration $c \subseteq F$ as $[\![M]\!]_c = \bigcup_{ac}\{M(ac) \mid c \models ac\}$. While product and configuration are often used synonymous in the literature, we adopt the distinction from the literature between those two elements and consider a configuration as an implementation-agnostic set of features whereas a product comprises the implementation generated for a configuration [28].

We denote all elements after evolution with a prime symbol (e.g., the feature model after evolution is F'). We define feature-model evolution using standard set operations. For instance, if the feature `GPS` is deleted, we express this as: $F' = F \setminus \{\texttt{GPS}\}$. As configurations are sets of selected features, we use common set operations to

[1] https://gitlab.com/DarwinSPL/GuyDance
[2] https://gitlab.com/mnieke/guydance_proofs

formalize update operations. For instance, we express the removal of the GPS feature from a configuration C with $C' = C \setminus \{\texttt{GPS}\}$.

To describe mapping evolution, we define a replace operator $M[f_n \mapsto exp]$ that iterates over all application conditions of M and replaces occurrences of a feature f_n by the feature expression exp. In the running example, if Diesel should be replaced by Engine in the mapping, we express this as $M' = M[\texttt{Diesel} \mapsto \texttt{Engine}]$. For simplicity and without loss of generality, we assume that if a realization artifact $i \in I$ is modified, this results in a new artifact $i' \in I'$. As new realization artifacts also require to be mapped to features, we define a second operator: $M \oplus (exp, i' \in I')$ adds an entry with the application condition exp related to the realization artifact i'.

We formalize product behavior and its preservation. As, in general, program behavior equality is undecidable [20], we rely on a more conservative notion for comparison. As approximation for product behavior, we use the definition of a product, i.e., $[\![M]\!]_C$. Product behavior of a configuration C is preserved if we can find a configuration C' that results in the same set of artifacts. Thus, we consider product behavior preservation as syntactic equality.

Definition 1. *For a product line (F, I, M) evolved to (F', I', M'), configurations $C \in \mathcal{P}(F)$, and $C' \in \mathcal{P}(F')$, the product behavior of C' preserves the product behavior of C, if*

$$[\![M]\!]_C = [\![M']\!]_{C'}$$

For instance, if feature GPS is mapped to $i_{\texttt{GPS}}$, feature Glonass is mapped to $i_{\texttt{Glonass}}$ and configuration $C = \{\texttt{GPS}, \texttt{Glonass}\}$ is used for product generation, the product behavior of C is defined by $[\![M]\!]_C = \{i_{\texttt{GPS}}, i_{\texttt{Glonass}}\}$. During evolution, Glonass is merged into GPS and $i_{\texttt{Glonass}}$ is mapped to GPS. By removing Glonass from C resulting in $C' = \{\texttt{GPS}\}$, C' preserves the product behavior of C (i.e., $[\![M]\!]_C = [\![M']\!]_{C'} = \{i_{\texttt{GPS}}, i_{\texttt{Glonass}}\}$).

To preserve product behavior of a configuration, this configuration may need to be updated. Note that a configuration is an implementation-agnostic set of features whereas a product comprises the implementation of a configuration. We identify configuration subsets that need to evolve by adapting the filter operator $\upharpoonright$ of Sampaio et al. [21]. For a feature model F and a feature expression exp, $F \upharpoonright exp$ yields the set of all configurations of F that satisfy exp. For instance, in the running example, if AudioCD and Cassette are deleted, all configurations that select both features need to be updated, i.e., the configuration yielded by $F \upharpoonright \texttt{AudioCD} \wedge \texttt{Cassette}$.

3 GUIDED CONFIGURATION EVOLUTION

Individual knowledge of neither domain engineers nor application engineers is sufficient to update configurations after product-line evolution. For instance, in the running example, domain engineers might not know the requirements of configurations that selected the Cassette feature and application engineers need to know that this feature has been deleted. We provide a methodology to support domain engineers in guiding application engineers on updating their configurations.

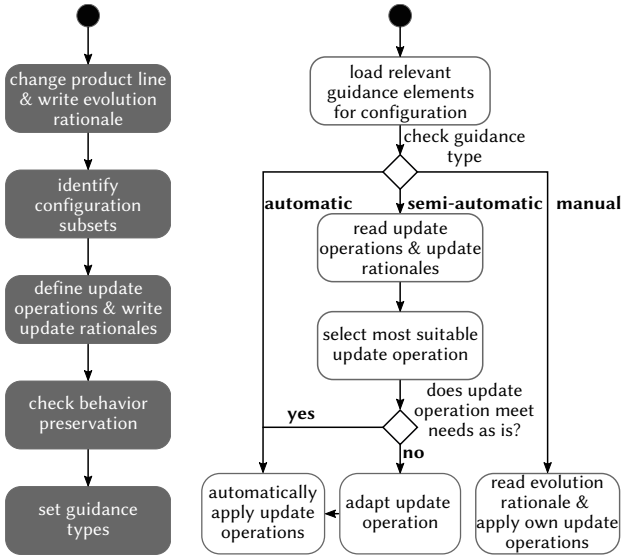
In such guidance, domain engineers formulate instructions for application engineers to update configurations in accordance with performed product-line evolution in a machine processable manner. Ideally, these instructions can be applied fully automatically and preserve a configuration's meaning in terms of product behavior – even if a different set of features has to be selected. However, in some cases, product behavior cannot be preserved by a new configuration and application engineers need to decide on which of the suggested configuration update operations to perform to find a configuration that best suites their use case. Application engineers can use guidance at a time of their choosing and independently of domain engineers to update configurations that are relevant to them. For a product line PL, application engineers derive a configuration C and a product represented by $[\![M]\!]_C$. After domain engineers change the product line to PL', they define guidance for application engineers to update their configuration to C' and corresponding product $[\![M']\!]_{C'}$, which can be derived from PL'. Depending of the intent of the evolution operation, the defined guidance may preserve product behavior. However, in all cases, domain engineers have to make clear whether product behavior is preserved, whether it is not preserved, or whether it is unknown. In particular, we conservatively assume that product behavior is not preserved if resulting products use different implementation artifacts than before evolution (cf. Section 2).

Table 1: Guidance for a *Delete Feature* operation.

	Operation: Delete feature f_0 with realization artifacts (r) $F' = F \setminus \{f_0\}$, $M' = M[f_0 \mapsto false]$				
	Configurations	**Update Operations**	**Preserves Behavior**	**Update Rationale**	**Type**
(x_1)	$\boldsymbol{Delete_0}$: $C \in F \upharpoonright \neg f_0$ (s_1)	($u_{1,1}$:) $C' = C$ ($op_{1,1}$)	yes ($b_{1,1}$)	Not affected. Can be left as-is. ($r_{1,1}$)	autom. (t_1)
(x_2)	$\boldsymbol{Delete_1}$: $C \in F \upharpoonright f_0$	$C' = C \setminus \{f_0\}$	no	Remove f_0 from all configs.	semi-autom.

3.1 Structure of Configuration Evolution Guidance

Configuration evolution guidance consists of the essence of product-line evolution, configuration update suggestions, and statements of product behavior preservation. Table 1 shows an example of guidance for a *Delete Feature* evolution operation. For easier reference, we added identifiers in brackets in the table which we refer to in the text. First, the rationale of the product-line evolution itself is defined in natural language (i.e., r in Table 1). This helps application engineers to understand the overall scope and reasons for changes that have been performed. Second, domain engineers have to define a set of guidance elements (i.e., $\mathcal{X}$). Each guidance element ($x_i \in \mathcal{X} = (s_i, \mathcal{U}_i, t_i)$, visualized as row in Table 1) is defined for a subset of configurations (s_i) for which it is applicable. As a result, domain engineers must not define an update operation for each individual configuration but can define one update operation for large subsets of configurations. Domain engineers define a set of configuration update operations ($\mathcal{U}_i$) for each guidance element. These update operations are suggestions for application engineers on how to update their configurations. For each update operation ($u_{i,j} \in \mathcal{U}_i$), domain engineers need to specify the concrete set operation on the configuration ($op_{i,j}$), a rationale ($r_{i,j}$) which explains why they defined this operation and in which cases it makes sense to be applied. Additionally, domain engineers specify whether product behavior is preserved ($b_{i,j}$) by applying a update operation (i.e., $u_{i,j} \in \mathcal{U}_i = (op_{i,j}, r_{i,j}, b_{i,j})$). In this way,



(a) For domain engineers. **(b) For application engineers.**

Figure 2: Guided configuration evolution process.

application engineers always know whether they have to perform additional work, e.g., testing updated products.

Finally, the type of a guidance element (t_i in Table 1) can be *automatic*, *semi-automatic*, or *manual*. Automatic guidance can be applied without any manual effort from application engineers. Semi-automatic guidance requires manual effort from application engineers in terms of choosing between multiple possible update operations that can be applied automatically. Manual guidance requires application engineers to specify update operations on their own using the information on the product-line evolution. This is required if domain engineers are not able to define update operations. Guidance $\mathcal{G} = (r, X)$ is defined as a tuple containing the evolution rationale and the set of guidance elements.

3.2 Guided Configuration Evolution Process

As part of the guided configuration evolution methodology, we propose processes for domain engineers to define guidance and for application engineers to apply such guidance. Figure 2a illustrates the process from the domain engineers' perspective. During evolution, domain engineers have to gradually define the elements for the configuration evolution guidance. First, they can perform product-line evolution as they are used to, e.g., with existing tools. To allow a high level of flexibility, this evolution is performed independently of our method and, consequently, we do not limit how to change a product line. Directly afterwards, so that no details on the evolution are forgotten, domain engineers define guidance. To share knowledge about the evolution, domain engineers have to specify an evolution rationale. The rationale should be specified in such a way that application engineers with different levels of expertise are able to understand it. Second, they have to determine which configuration subsets are affected by the product-line evolution. This is done by analyzing which features are part of the evolution scenario. For instance, if the feature `Cassette` of Figure 1 is deleted during evolution, all configurations selecting `Cassette` are in one category and all configurations not selecting `Cassette` are in another category. Third, for each of these subsets, one or multiple update operations should be defined and rationales explaining them with their impact on configurations must be added. Multiple update operations are necessary if the domain engineer identifies several sensible possibilities to update those configurations. If domain engineers are not able or do not want to define update operations, we allow to omit the respective update operations which results in manual typed guidance for application engineers. However, this is an undisciplined usage of our method, and we strongly encourage domain engineers to define update operations.

Fourth, domain engineers have to analyze how each update operation affects product behavior. Given that our behavior preservation notion is based on the set of artifacts included in a product, this is optimally done with tool support, e.g., with a verification system that compares the resulting artifacts of a configuration by evaluating the mapping before and after evolution. Different levels of product behavior assurance may be defined by domain engineers. For instance, *proven* if product behavior preservation is shown using a proof system, *tested* if thorough testing resulted in the same product behavior, or *reviewed* if experts reviewed the resulting product and confirm product behavior preservation. Fifth, a guidance type has to be set for each update operation, determining the automation degree of the guidance. For cases in which the update operation is clear, domain engineers set the type to *automatic*, e.g., if the evolution was a refactoring or if only one operation is possible. However, this type should only be used if product behavior is preserved or if other circumstances force this operation (e.g., management decisions). If multiple update operations are available or if domain engineers are not sure whether the update operation is suitable, the type is set to *semi-automatic*. We consider it as undisciplined usage if no update operation has been defined, and set the type to *manual*.

Figure 2b shows the process from the application engineers' perspective. When application engineers want to update a configuration to a new product-line version the knowledge transfer takes place. It is first evaluated which guidance type is set for that configuration. If the category is *automatic*, the respective update operation can be applied automatically, without manual effort from application engineers. Nevertheless, the update operation and the rationale can still be inspected by application engineers. If the category is *semi-automatic*, application engineers have to select the most suitable update operation based on the rationales. The selected operation can then be applied automatically. Moreover, application engineers can adapt the update operations if needed. If it is *manual*, the application engineers can read the rationales that explain the product-line evolution. Based on this information, application engineers need to find a fix on their own.

4 GUIDANCE TEMPLATES

Specifying guidance for product-line evolution requires up-front effort. Thus, to reduce effort for domain engineers, we provide the possibility to store guidance for evolution scenarios in the form of

Table 2: Template *Merge Features*

Operation: Merge functionality of feature f_1 into feature f_0 $F' = F \setminus \{f_1\}$, $M' = M[f_1 \mapsto f_0]$			
Configurations	**Update Operations**	**Preserves Behavior**	**Type**
$Merge_0$: $C \in F \upharpoonright (\neg f_0 \wedge \neg f_1)$	$C' = C$	yes	autom.
$Merge_1$: $C \in F \upharpoonright (f_0 \wedge f_1)$	$C' = C \setminus \{f_1\}$	yes	autom.
$Merge_2$: $C \in F \upharpoonright (f_0 \wedge \neg f_1)$	$M_{2.a}$: $C' = C$	no	semi-autom.
	$M_{2.b}$: $C' = C \setminus \{f_0\}$		
$Merge_3$: $C \in F \upharpoonright (\neg f_0 \wedge f_1)$	$M_{3.a}$: $C' = C \setminus \{f_1\}$	no	semi-autom.
	$M_{3.b}$: $C' = C \setminus \{f_1\} \cup \{f_0\}$		

Table 3: Template *Extract New Feature*

Operation: Extract some functionality of feature f_0 into new feature f_1 $F' = F \cup \{f_1\}$, $M' \subseteq \{m' = m, m' = m[f_0 \mapsto f_1], m' = m[f_0 \mapsto (f_0 \wedge f_1)], m' = m[f_0 \mapsto (f_0 \vee f_1)] \mid m \in M\}$			
Configurations	**Update Operations**	**Preserves Behavior**	**Type**
$Extract_0$: $C \in F \upharpoonright \neg f_0$	$E_{0.a}$: $C' = C$	yes	semi-autom.
	$E_{0.b}$: $C' = C \cup \{f_0\}$	no	
	$E_{0.c}$: $C' = C \cup \{f_1\}$		
$Extract_1$: $C \in F \upharpoonright f_0$	$E_{1.a}$: $C' = C \cup \{f_1\}$	yes	semi-autom.
	$E_{1.b}$: $C' = C$	no	
	$E_{1.c}$: $C' = C \setminus \{f_0\} \cup \{f_1\}$		

templates to facilitate reuse. Consequently, guidance templates further automate the presented process, but are not necessary to apply our method. In contrast to "standard" guidance, templates additionally specify the evolution scenario for which they are applicable. An evolution scenario $\mathcal{E} = (e_F, e_M)$ consists of feature-model evolution (e_F) and mapping evolution (e_M), described in terms of the evolution operations we defined in Section 2. The evolution operations are preconditions for applying the guidance defined in the templates. Thus, an evolution template $\mathcal{T} = (\mathcal{G}, \mathcal{E})$ consists of a description of the evolution scenario and the corresponding guidance.

In the following, we define three exemplary guidance templates for common evolution scenarios. We chose those scenarios as related work identified them as relevant evolution cases [13, 15, 17, 21]. The templates also illustrate the general concept of guided configuration evolution. For brevity, we omit the rationales in the tables describing the templates but explain them in the text. To better reference elements of the table in the text, we add identifiers for guidance elements and update operations.

4.1 Delete Feature

Maintaining certain features may not be profitable anymore. In the running example (cf. Figure 1), the feature `Cassette` is rarely used. Therefore, this feature is deleted, including its mapped artifacts. For such cases, we introduce the *Delete Feature* template.

We use this template to illustrate the structure of guided configuration evolution with Table 1. As precondition, the feature f_0 is removed from the feature set and in the mapping application conditions, it is replaced by *false*. The first guidance element ($Delete_0$) addresses the configuration subset not selecting f_0. We specify the category as *automatic* because such configurations remain unchanged, as they are unaffected by the operation, and explain this in the rationale.

We define a second guidance element $Delete_1$ for the configuration subset selecting f_0. As update operation, we specify to remove f_0 from these configurations and state in the update rationale that we suggest this as f_0 no longer exists. As product behavior is not preserved if artifacts were mapped to f_0 before evolution, we set the guidance category to *semi-automatic* as application engineers should be informed of the reduced functionality. Nonetheless, this update operation can be applied automatically.

4.2 Merge Features

When systems evolve, individual features may grow together into one semantic unit [17]. In our running example, new cheaper hardware is capable of providing functionality for both features `GPS` and `Glonass`. Thus, `Glonass` is merged into `GPS`. For such cases, we define the *Merge Features* template.

Table 2 shows this template. The *source* feature f_1 is merged into the *target* feature f_0 and, thus, f_1 is removed from the set of features and f_1 is replaced by f_0 in all mapping application conditions.

We define four guidance elements for this template. The first element $Merge_0$ is for the configuration subset selecting neither f_0 nor f_1. As the merge does not affect them, we leave the configurations unchanged. Thus, product behavior is preserved, no interaction is required, and we set the guidance category to *automatic*. We define the second guidance element $Merge_1$ for the configuration subset which selects both f_0 and f_1. As update operation, we specify to remove f_1 as f_0 provides functionality for both features after evolution. Product behavior is preserved using this update operation and, thus, we set the guidance category to *automatic*.

The third guidance element $Merge_2$ is for configurations containing f_0 but not f_1. In this case, existing approaches [30–32] detecting defects in configurations would leave the configuration as-is because f_0 still exists. As f_0 also provides the functionality of f_1, we know that product behavior is *not* preserved. In the first update operation $M_{2.a}$, we define that the configuration is left as-is but we make application engineers aware that product behavior changed. Without this knowledge, products with altered behavior might be deployed which may cause harm. We provide a second update operation $M_{2.b}$ that removes f_0 from configurations if application engineers do not want to have the additional functionality of f_1.

As we do not know which update operation is most suitable for application engineers, the guidance type is *semi-automatic*. Thus, application engineers must select an update operation that can be applied automatically. $Merge_3$ describes the remaining case and is defined analogously to $Merge_2$.

4.3 Extract New Feature

Features represent a cohesive unit of configuration. To allow more precise configuration, parts of a feature's functionality can be extracted into a separate feature. In our running example, both engine types are equipped with a turbocharger and this functionality is integrated into both features. For cheaper variants, the turbocharger should be optional. Thus, this functionality is extracted

into a new feature `Turbocharger`. For such cases, we introduce the *Extract New Feature* template shifting functionality from a *source* feature into a new *target* feature.

Table 3 shows this guidance template. We add a new feature f_1 to the feature set. As some artifacts mapped to f_0 should be extracted to f_1, we need to represent this in the mapping. We identified four cases: first, if an artifact remains mapped to f_0 after evolution, we leave the mapping as-is; second, if an artifact belongs to the functionality that is extracted, we replace f_0 by f_1 in the application condition; third, if an artifact is required only to make both features work together, we replace f_0 by $f_0 \wedge f_1$ in the application condition; fourth, if an artifact is required by both features individually, we replace f_0 by $f_0 \vee f_1$ in the application condition. As the required evolution operation may differ for each artifact, domain engineers can change each application condition independently.

We define guidance elements for two configuration subsets. First, $Extract_0$ targets configurations not selecting f_0. Principally, those configurations could be left as-is and product behavior would be preserved. However, new configuration options are introduced and application engineers might want to use them. Consequently, we define three update operations. The first update operation $\mathcal{E}_{0.a}$ leaves corresponding configurations unchanged and preserves product behavior. The configuration update operations $\mathcal{E}_{0.b}$ and $\mathcal{E}_{0.c}$ add f_0 or f_1, respectively. The two latter update operations do not preserve product behavior. To make application engineers aware of these new configuration options, we set the guidance type to *semi-automatic*.

The second guidance element $Extract_1$ targets subsets of configurations that select f_0. Again, product behavior could be preserved by adding f_1 to these configurations. Similar to $Extract_0$, application engineers might want to use the new configuration options. Consequently, we define three update operations. The first operation $\mathcal{E}_{1.a}$ adds the feature f_1 to the configurations as described above. The second operation $\mathcal{E}_{1.b}$ leaves the configuration as-is. The resulting product's functionality is reduced by the extracted functionality of f_1. The third operation $\mathcal{E}_{1.c}$ is relevant only if the functionality that has been extracted should be available. Correspondingly, f_0 is replaced by f_1 in configurations. Again, the latter two operations result in altered product behavior.

For this evolution scenario, existing approaches fixing defects in configurations [30–32] would leave the configuration as-is because f_0 still exists. In configurations covered by $Extract_0$ this would even preserve product behavior but application engineers would not be informed about the new configuration options. However, for configurations covered by $Extract_1$ this would even lead to changed product behavior which may entail significant risk and cost to later fix and update these configurations.

4.4 Evolution Process with Templates

The three presented templates are examples that illustrate the usage of guided configuration evolution, and we do not claim completeness. Hence, as additional templates may be necessary, we enable domain engineers to define their own templates. However, our methodology can also be applied without templates following the process defined in Section 3.2.

To cover guided configuration evolution with and without templates, we need to adapt the process defined in Section 3.2. After selecting a template to be applied, domain engineers apply the defined feature-model and mapping evolution operations. As the feature-model and mapping evolution operations defined in the templates are preconditions for applying the template, template's operations have to match the actually performed changes to the product line.

In the following steps, domain engineers have to check whether the elements defined in the template meet their needs. Optimally, the update operations meet the needs as-is and it is not necessary to change the update operations. However, domain engineers should always check whether they can define additional domain-specific update operations to better guide application engineers. If the update operations are not completely matching the evolution scenario or the intended way to update configurations, existing update operations can be adapted or supplemented by additional operations. For instance, if a feature should be replaced by another feature, the delete feature template can be applied with the first feature to be deleted but domain engineers can adapt the update operations such that the first feature is replaced by the latter feature in configurations. For changed or added update operations, domain engineers need to analyze whether product behavior is preserved. If domain engineers claim product behavior preservation for new update operations, they have to ensure that the resulting artifact set is the same afterwards, e.g., with a formal proof or excessive testing. In the next step, the guidance types of the guidance elements should be set. To stimulate domain engineers in providing more information, we define this as a mandatory step. Finally, the rationales for the evolution operation and the update operations should be written. This is of particular importance as application engineers should use this information as main source for decision making on how to update configurations.

By using templates, we expect that the effort for defining guidance can be reduced. If a template can be used as-is, the effort is almost non-existent. Adapted or newly defined templates can be added to a template catalog and, over the entire life cycle of a product line, the template catalog can grow to cover most evolution scenarios. Additionally, as the product-line evolution is formally specified in the templates, our methodology lies the foundation for an automated detection of evolution scenarios and, thus, suitable templates. Such an automated detection would reduce the effort for domain engineers even more as they do not have to search for an applicable template. Even if effort remains unchanged, it results in proactively avoiding errors instead of retroactively fixing errors constituting a quality assurance mechanism. This process shows the flexibility of the guided configuration evolution as it can be used from scratch without any templates, it can be gradually extended by templates, existing templates can be reused directly, or existing templates can be adapted for a concrete scenario.

5 APPLYING GUIDED CONFIGURATION EVOLUTION

Tool support is pivotal for using guided configuration evolution in real-world development projects. Thus, we sketch the core functions a production tool needs to provide based on our methodology along with the suitable application orders of these functionalities

to realize the workflows of Figures 2a, and 2b. We implemented an early open-source prototype, named *GuyDance*, to show feasibility (cf. Footnote 1).

Preserving compatibility with existing processes and tools is crucial for acceptance. For this reason, domain engineers can perform changes to their product line with tools they are used to. This is particularly important as guided configuration evolution can be used for certain important evolution operations but does not have to be used for all operations. Thus, if domain engineers consider a change as insignificant, our method does not have to be applied but it can be applied for other changes or even retroactively when first problems occur. In the next step, domain engineers have three options. First, they can define guidance without using an existing template. Second, they can define guidance and save this guidance as a new template. Third, they can reuse an existing template with or without adaptation. To define guidance and respective templates, a domain-specific language that provides the possibility to specify respective information is most suitable. In *GuyDance*, we used Xtext[3] for defining a grammar and editors for guidance and templates.

To increase the level of automation, templates that match the changes performed by domain engineers could be automatically detected by analyzing the actually performed changes and comparing to the changes defined in the templates. For instance, the tool FEVER [8] is able to extract and detect changes that match a certain pattern, such as evolution scenarios described in the templates.

For application engineers, the first step is to analyze which guidance elements (i.e., rows in the example tables) are relevant for an existing configuration. As the configuration subset of a guidance element is defined formally, this can be evaluated using a SAT solver. For instance, if a subset is defined as $C \in F \upharpoonright \neg f_0$ and a configuration selects features f_1, f_2, a SAT solver or simple Boolean evaluation algorithm can check the formula $\neg f_0 \wedge f_1 \wedge f_2$ for satisfiability. In this example, the configuration would be part of the defined subset, i.e., the guidance element is relevant. Next, update operations for the configuration are selected for application. If the guidance type is *automatic*, this can be done without user interaction. Nonetheless, a tool should give the possibility for application engineers to inspect this and to intervene if needed. For *semi-automatic* guidance, application engineers have to select which update operation to apply. To increase user experience, the effect of these operations can be shown as a preview. The actual execution of the update operations can be fully automated as the update operations are defined as set operations on configurations. To apply a update operation, selected features of an existing configuration are either deselected or newly selected features are added to that configuration.

6 EVALUATION

Knowledge whether product behavior is preserved after applying update operations is a core information of guided configuration evolution. In our evaluation, we show the soundness of our methodology by formally proving that the guidance we provide for the evolution templates (cf. Section 4) preserves behavior for the respective configuration subsets.

For our proofs, we utilize the formalization that we introduced in Section 2, i.e., product behavior of a configuration C is preserved by C', if $[\![M]\!]_C = [\![M']\!]_{C'}$. We fully formalized proofs for the three templates using the theorem prover *PVS* [16]. To this end, we formalized the evolution operations and the update operations in *PVS*. For the sake of brevity, we only provide proof sketches. The complete proofs can be found in our online repository (cf. Footnote 2).

For the *Delete Feature* template, behavior is preserved for configurations that did not select the deleted feature f_0 (cf. Table 1, $Delete_0$). To show this, we prove the following theorem.

Theorem 1. *For product line (F, I, M) evolved to (F', I', M'), given that $I \subseteq I'$, $f \in F, F' = F \setminus \{f\}$ and $M' = M[f \mapsto false]$:*

$$\forall C \in F \upharpoonright (\neg f) : [\![M]\!]^I_C = [\![M']\!]^{I'}_C$$

The idea of the proof is that we can show for an arbitrary M, an $M' = M[f \mapsto false]$ exists which produces the same value for configurations not containing f. We have proven this in *PVS* by induction over the application conditions of the mapping. We have proven all of the following theorems in *PVS* using similar reasoning.

For the *Merge Features* template, behavior is preserved if either both features f_0 and f_1 were not selected in C or both features were selected (cf. Table 2, $Merge_0$ and $Merge_1$). In the first case, the configuration remains as is and, in the second, f_1 is removed. To show behavior preservation for $Merge_0$ and $Merge_1$, we have proven the following theorem in *PVS*:

Theorem 2. *For product line (F, I, M) evolved to (F', I', M'), given that $I \subseteq I'$, with $f_0, f_1 \in F, f_0 \neq f_1, F' = F \setminus \{f_1\}$, and $M' = M[f_1 \mapsto f_0]$:*

$$(\forall C \in F \upharpoonright (\neg f_0 \wedge \neg f_1) : C' = C, [\![M]\!]^I_C = [\![M']\!]^{I'}_{C'}) \wedge$$
$$(\forall C \in F \upharpoonright (f_0 \wedge f_1) : C' = C \setminus \{f_1\}, [\![M]\!]^I_C = [\![M']\!]^{I'}_{C'})$$

For the *Extract New Feature* template, we are principally able to preserve behavior for all possible configurations. In particular, for configurations that do not select the feature f_0, we leave the configuration as is, and for configurations that select f_0, we additionally select the extracted feature f_1 (cf. Table 3, $Extract_{0.a}$ and $Extract_{1.a}$). In *PVS*, we formalized and proved the template using the following theorem:

Theorem 3. *For product line (F, I, M) evolved to (F', I', M'), given that $I \subseteq I'$, with $f_0 \in F, f_0 \neq f_1, F' = F \cup \{f_1\}$ and $M' \subseteq \{m' = m, m' = m[f_0 \mapsto f_1], m' = m[f_0 \mapsto (f_0 \wedge f_1)], m' = m[f_0 \mapsto (f_0 \vee f_1)] \mid m \in M\}$:*

$$(\forall C \in F \upharpoonright (\neg f_0) : C' = C, [\![M]\!]^I_C = [\![M']\!]^{I'}_{C'}) \wedge$$
$$(\forall C \in F \upharpoonright (f_0) : C' = C \cup \{f_1\}, [\![M]\!]^I_C = [\![M']\!]^{I'}_{C'})$$

Threats to Validity. The external validity is threatened as we prove behavior preservation for only three templates. However, it is infeasible to consider all possible evolution operations. All the more so, as the templates' update operations may vary between product lines to match domain-specific needs. We try to mitigate this threat by defining templates for evolution operations considered as relevant in the literature [13, 15, 17, 21]. We expect that other evolution operations work similarly.

Another threat to validity is that we do not evaluate the definition of new guidance and templates, especially for real-world product-line evolution. As we defined the three templates in this paper using our methodology, we expect that defining new templates and proving behavior works analogously to the definition of the

[3] https://www.eclipse.org/Xtext/

presented templates. In our future work we want to define guidance for a real-world product-line evolution scenario.

One of the main goals of our methodology is to automate updating configurations for application engineers and to provide support in case of altered product behavior. However, we did not evaluate to which extent we are able to preserve product behavior and detect altered product behavior. We want to investigate this in our future work for real-world product-line evolution.

7 RELATED WORK

Xu et al. [33] identified misconfigurations in highly configurable systems that lead to vulnerabilities or bugs. They conclude that developers should take an active role in handling misconfigurations by supporting users in the configuration process. With our methodology, we address this issue as we provide a method for domain engineers (i.e., developers) to support application engineers (i.e., users). Zhang et al. [34] address a very similar problem as guided configuration evolution. They are interested in preserving product behavior after evolution by analyzing products' control flow. This method could be used complementarily by domain engineers if product behavior cannot be preserved to devise a suggestion for a update operation.

Recent research analyzed and categorized evolution of product lines and, in particular, the mapping between variability model and artifacts [6, 8, 17, 35]. However, the guided configuration evolution is more generic and helps to update configurations. With FEVER, Dintzner et al. introduced a tool to extract changes to variability models, code artifacts, and the corresponding mapping [8]. FEVER could be used in combination with our methodology to identify commits of a product line that match a certain pattern, such as the evolution scenarios described with guidance templates.

Other research defines refactorings for product-line evolution. Thüm et al. [27] and Alves et al. [1] classify feature-model evolution in terms of changes to the set of valid configurations. Both approaches do not consider product behavior of configurations. Schulze et al. define refactoring operations for product lines using feature-oriented and delta-oriented programming [24, 25]. Seidl et al. define evolution operations to co-evolve three spaces: feature models, artifacts, and mappings [26]. For operations affecting more than one space, they define how to co-evolve the other spaces [26]. In contrast to the previously mentioned publications, we do not want to limit evolution to refactorings.

Borba et al. devised a refinement theory for product-line evolution preserving product behavior [4] and Neves et al. proposed several evolution templates preserving product behavior using this theory [13]. Sampaio et al. extended this theory by introducing partially safe evolution templates, preserving product behavior for a subset of configurations [21, 22]. These methods already allow to reason on product behavior changes of configurations even in presence of configuration changes. We devised a novel more general concept that enables domain and application engineers can share their knowledge to update configurations after product-line evolution. Thus, domain-specific knowledge can be incorporated and guidance can also be provided even if product behavior cannot be preserved. We used the formalizations and proofs of the works of Borba et al. [4], Neves et al. [13], and Sampaio et al. [21, 22] as a basis for our formalization and the proofs for the templates.

Some research focuses on fixing invalid configurations. An automatic approach computes the smallest possible set of changes in the configuration to fix it [31]. Semi-automatic approaches proposed either to provide the complete set of fixes with the smallest amount of feature changes [32] or to gradually reach the desired fix using application engineers' feedback [30]. Both semi-automatic approaches assume that the person fixing the configuration knows what the best fix is. Moreover, these approaches do not take the implementation and mapping into account. Thus, the fixes may lead to different product behavior and, therefore, provide a false sense of correctness.

8 CONCLUSION

We presented guided configuration evolution, a methodology for updating configurations after product-line evolution that overcomes the communication barrier between domain engineers and application engineers. We enable domain engineers to share the essence of product-line evolution and to suggest configuration update operations. Application engineers can use this information to update their configurations while knowing the impact on product behavior. Even if it is impossible to talk, our methodology allows for application engineers to update configurations in accordance with the evolution performed by domain engineers, at the time of their choosing, and with the most suitable update strategy. Additionally, effort is spent only once by domain engineers to define guidance which can be used by an arbitrary number of application engineers, optimally resulting in a reduced overall effort.

This work raises several further research opportunities. First and most importantly, we lay the theoretical and practical foundations for guided configuration evolution. To assess effectiveness, efficiency, and acceptance for real-world product-line evolution processes, we plan to perform an empirical evaluation based on real-world product lines and their evolution. A second future work opportunity is an extension to our method that ensures configuration validity after applying update operations, which would reduce manual effort of application engineers even more. Third, we want to investigate automatic learning from modified templates (either by domain or by application engineers) to derive new templates or to sustainable change templates. Finally, if domain engineers define their own templates, automatic proofs of behavior preservation would increase usability, as proofs in *PVS* are typically not feasible for them.

ACKNOWLEDGMENT

This work was partially supported by the Federal Ministry of Education and Research of Germany within CrESt (funding 01IS16043S), by the DFG (German Research Foundation) under SPP1593: Design For Future — Managed Software Evolution, by FACEPE (grant APQ-0570-1.03/14), by CNPq (grant 409335/2016-9), and by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0. Sampaio was supported by a CAPES Foundation Scholarship, process number 88881.129599/2016-0.

REFERENCES

[1] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. 2006. Refactoring product lines. In *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*. 201–210. https://doi.org/10.1145/1173706.1173737

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-oriented software product lines: concepts and implementation*. Springer Science & Business Media.

[3] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *Model-Driven Engineering Languages and Systems*, Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran (Eds.). Springer International Publishing, Cham, 302–319.

[4] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A theory of software product line refinement. *Theoretical Computer Science* 455 (2012), 2–30.

[5] J. Bosch. 2001. Software product lines: organizational alternatives. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* 91–100. https://doi.org/10.1109/ICSE.2001.919084

[6] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2016. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering* 23, 4 (2016), 687–733.

[7] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. 2000. Generative programming and active libraries. In *Generic Programming*. Springer, 25–39.

[8] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2017. FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering* (04 Nov 2017). https://doi.org/10.1007/s10664-017-9557-6

[9] Karine Gomes, Leopoldo Teixeira, Thayonara Alves, Márcio Ribeiro, and Rohit Gheyi. 2019. Characterizing Safe and Partially Safe Evolution Scenarios in Product Lines: An Empirical Study. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems* (Leuven, Belgium) *(VAMOS '19)*. Association for Computing Machinery, New York, NY, USA, Article Article 15, 9 pages. https://doi.org/10.1145/3302333.3302346

[10] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[11] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. Identifying the Intensity of Variability Changes in Software Product Line Evolution. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3233027.3233032

[12] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 155–166. https://doi.org/10.1145/3238147.3238201

[13] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demóstenes Sena, and Uirá Kulesza. 2015. Safe evolution templates for software product lines. *Journal of Systems and Software* 106 (2015), 42–58. https://doi.org/10.1016/j.jss.2015.04.024

[14] Michael Nieke, Gil Engel, and Christoph Seidl. 2017. DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-aware Software Product Lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems* (Eindhoven, Netherlands) *(VAMOS '17)*. ACM, New York, NY, USA, 92–99. https://doi.org/10.1145/3023956.3023962

[15] Michael Nieke, Christoph Seidl, and Sven Schuster. 2016. Guaranteeing Configuration Validity in Evolving Software Product Lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems* (Salvador, Brazil) *(VaMoS '16)*. ACM, New York, NY, USA, 73–80. https://doi.org/10.1145/2866614.2866625

[16] Sam Owre, John M Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *International Conference on Automated Deduction*. Springer, 748–752.

[17] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of variability models and related software artifacts. *Empirical Software Engineering* 21, 4 (2016), 1744–1793. https://doi.org/10.1007/s10664-015-9364-x

[18] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. ACM, New York, NY, USA, 78–83. https://doi.org/10.1145/3336294.3336322

[19] K. Pohl, G. Böckle, and F.J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Berlin Heidelberg.

[20] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[21] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2016. Partially Safe Evolution of Software Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference* (Beijing, China) *(SPLC '16)*. ACM, New York, NY, USA, 124–133. https://doi.org/10.1145/2934466.2934482

[22] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2019. Partially safe evolution of software product lines. *Journal of Systems and Software* 155 (2019), 17 – 42. https://doi.org/10.1016/j.jss.2019.04.051

[23] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (01 Oct 2012), 477–495. https://doi.org/10.1007/s10009-012-0253-y

[24] Sandro Schulze, Oliver Richers, and Ina Schaefer. 2013. Refactoring Delta-oriented Software Product Lines. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development* (Fukuoka, Japan) *(AOSD '13)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2451436.2451446

[25] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. 2012. Variant-preserving Refactoring in Feature-oriented Software Product Lines. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems* (Leipzig, Germany) *(VaMoS '12)*. ACM, New York, NY, USA, 73–81. https://doi.org/10.1145/2110147.2110156

[26] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. 2012. Co-evolution of Models and Feature Mapping in Software Product Lines. In *Proc. of the 16th Intl. Software Product Line Conference* (Salvador, Brazil) *(SPLC '12)*. ACM, New York, NY, USA, 76–85. https://doi.org/10.1145/2362536.2362550

[27] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning About Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 254–264. https://doi.org/10.1109/ICSE.2009.5070526

[28] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proceedings of the 2011 15th International Software Product Line Conference (SPLC '11)*. IEEE Computer Society, Washington, DC, USA, 191–200. https://doi.org/10.1109/SPLC.2011.53

[29] Thomas Thüm, Sebastian Krieter, and Ina Schaefer. 2018. Product Configuration in the Wild: Strategies for Conflicting Decisions in Web Configurators. In *Proceedings of the 20th Configuration Workshop, Graz, Austria, September 27-28, 2018*. 1–8.

[30] Bo Wang, Leonardo Passos, Yingfei Xiong, Krzysztof Czarnecki, Haiyan Zhao, and Wei Zhang. 2013. SmartFixer: Fixing Software Configurations Based on Dynamic Priorities. In *Proceedings of the 17th International Software Product Line Conference* (Tokyo, Japan) *(SPLC '13)*. ACM, New York, NY, USA, 82–90. https://doi.org/10.1145/2491627.2491640

[31] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. 2008. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *2008 12th International Software Product Line Conference*. 225–234. https://doi.org/10.1109/SPLC.2008.16

[32] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. 2012. Generating Range Fixes for Software Configuration. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 58–68. http://dl.acm.org/citation.cfm?id=2337223.2337231

[33] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 244–259. https://doi.org/10.1145/2517349.2522727

[34] Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 152–163. https://doi.org/10.1145/2568225.2568251

[35] Andreas Ziegler, Valentin Rothberg, and Daniel Lohmann. 2016. Analyzing the Impact of Feature Changes in Linux. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems* (Salvador, Brazil) *(VaMoS '16)*. ACM, New York, NY, USA, 25–32. https://doi.org/10.1145/2866614.2866618