Software and Systems Modeling
https://doi.org/10.1007/s10270-021-00906-w

REGULAR PAPER

Guiding the evolution of product-line configurations

Michael Nieke[1] · Gabriela Sampaio[2] · Thomas Thüm[3] · Christoph Seidl[4] · Leopoldo Teixeira[5] · Ina Schaefer[1]

Received: 29 September 2020 / Revised: 7 May 2021 / Accepted: 14 June 2021

Abstract
A product line is an approach for systematically managing configuration options of customizable systems, usually by means of features. Products are generated for 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 interact with 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 automatic update operations for configurations. As an effect, we enable knowledge transfer between those engineers without the need for interactions. We evaluate our methodology on four large-scale industrial product lines. The results of the qualitative evaluation indicate that our method is flexible enough for real-world product-line evolution. The quantitative evaluation indicates that we detect product behavior changes for up to 55.3% of the configurations which would not have been detected using existing methods.

Keywords Product line · Product-line evolution · Guided feature configuration evolution · Product behavior preservation

1 Introduction

Configurable software allows customization of software products to fit user requirements. For instance, cars and their software can be configured by customers through a web configurator of the car manufacturer [1] and the Linux kernel can be custom-tailored by selecting from more than 21,000 configuration options [2]. A product line is a concept for managing configurable software systems and their configuration options in terms of features [3–6]. A *configuration* of a product line is a set of selected features. The set of available features is captured using a *feature model* [5]. Features are associated with reusable artifacts or parts thereof by Boolean formulas in a *feature-artifact mapping* (e.g., through preprocessor statements in C++ code). Using these artifacts, a *product* can be generated automatically for a given configuration [4,7]. In the product-line life cycle, two main roles are involved: *domain engineers* specify feature models and feature-artifact mappings [3]; *application engineers* define configurations to generate products.

Product-line evolution is a well-acknowledged field and current field of research [8–11]. Similar to all other software systems, evolution of product lines is ubiquitous due to changed or new requirements. In the process of product-line evolution, domain engineers may change the feature model, artifacts, and the feature-artifact mapping [12]. This can lead to unintended changes to product behavior [13]. For instance, if a feature `A` is merged into another feature `B`, configurations

Communicated by Hong Mei.

✉ Michael Nieke
micni@itu.dk

Gabriela Sampaio
g.sampaio17@imperial.ac.uk

Thomas Thüm
thomas.thuem@uni-ulm.de

Christoph Seidl
chse@itu.dk

Leopoldo Teixeira
lmt@cin.ufpe.br

Ina Schaefer
i.schaefer@tu-bs.de

1 TU Braunschweig, Brunswick, Germany

2 Imperial College London, London, United Kingdom

3 Ulm University, Ulm, Germany

4 ITU Copenhagen, Copenhagen, Denmark

5 Federal University of Pernambuco, Recife, Brazil

Published online: 04 July 2021

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 [14,15]. Thus, configuration evolution must be in line with product-line evolution. Up to now, application engineers are left with the task of manually detecting and fixing problems with their configurations used in the field, which is time consuming and error prone [16].

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 [17]. For instance, a developer of the Linux kernel (domain engineer) does not generally know all end-users (application engineers), their specific configurations, or requirements which led them to a specific feature selection. Hence, a domain engineer in isolation cannot give direct advice to application engineers on how to update their configurations after evolution.

Application engineers can know when evolution was performed, by means of patches or changelogs. Nonetheless, they still lack information regarding the why and how, that is, the rationale for the particular changes that were performed. This makes it difficult to 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 [18,19]. Previous research provides automated fixes for invalid configurations [16,19–21]. However, these approaches fix configurations only on a syntactical level but not on a semantical level, which may inadvertently alter product behavior. Apart from the lack of product behavior consideration, these approaches assume that engineers in isolation are able to choose a suitable fix. In summary, 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 communicate with each application engineer [17]. For instance, thousands of application engineers configure the Linux kernel and hundreds of domain engineers develop it. To update configurations, the domain engineers would need to communicate with each application engineer. Moreover, industry reports that, for some systems, configuration logic changes almost every week [14]. 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*, a methodology to provide guidance for updating configurations to a new product-line version. Our main goal is to automate the process of updating configurations semantically as far as possible. We acknowledge the deficits in the individual engineer's knowledge and barriers for communication and provide a concept to encode necessary information in a machine and human-processable format. 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 that is used by application engineers to update their configurations. Guidance defined by a domain engineer 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 for each evolution step and can be reused by an unlimited number of application engineers. In addition, domain engineers do not have to define guidance for each individual configuration, but can define it symbolically for large sets of configurations. While existing work considers individual engineers to be able to update configurations [16,19–22], we identify the communication barrier between domain and application engineers as a central challenge. With our methodology this communication barrier does not become a problem. We argue that the first step to overcome this barrier is to focus on product behavior instead of configuration validity. 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 specify how the set of features and the feature-artifact mapping evolve. We perform three evaluations: First, we formally prove that we are able to preserve product behavior of configurations for typical evolution scenarios using our methodology. Second, in a qualitative evaluation, we analyze whether it is feasible to apply and adapt our methodology. Third, in a quantitative evaluation, we analyze the percentage of configurations to which we can (a) apply guidance automatically, (b) preserve product behavior, and (c) detect unnoticed changes to product behavior. The quantitative evaluation is split into two parts: first, we use real-world

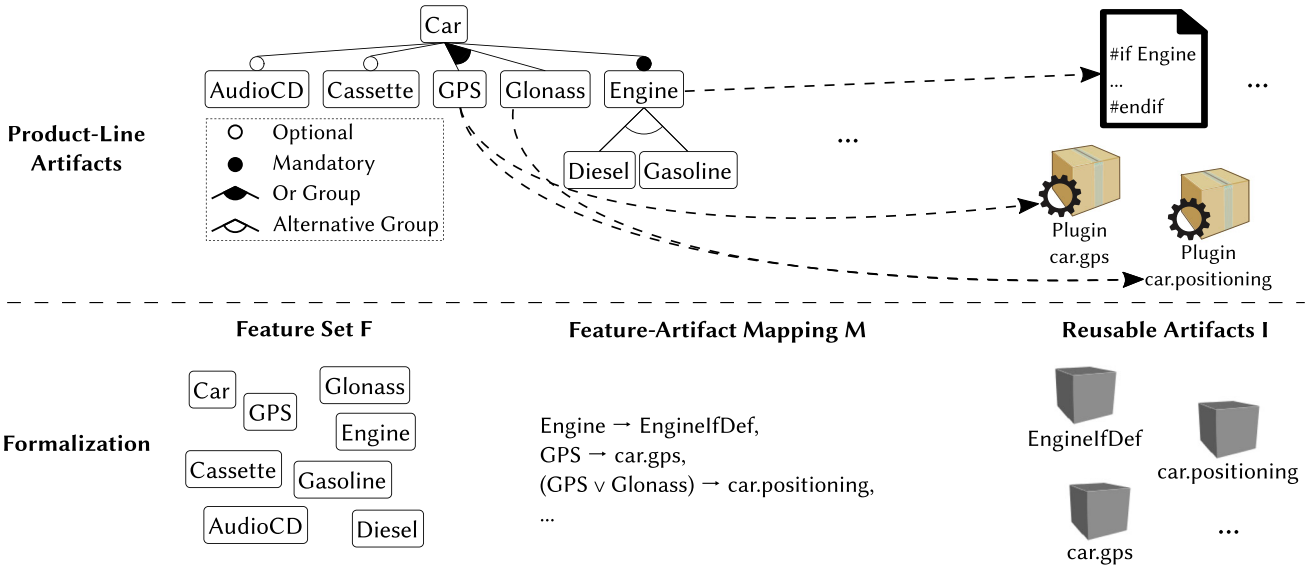


Fig. 1 Artifacts of an exemplary car product line and their formalization of the running example

feature-model evolution and evaluate the domain engineer's perspective; second, we use two real-world feature models with existing configurations and evaluate the application engineer's perspective. In summary, we make the following contributions:

1. We propose a formal foundation for domain engineers to express evolutionary changes to configurations.
2. We define a methodology with a prototypical tool. GuyDance[1] enabling domain engineers to guide application engineers in updating configurations.
3. We provide three example evolution templates to support domain engineers, which illustrate the methodology and the formalism.
4. We formally prove soundness of the templates by establishing behavior preservation for subsets of configurations.[2]
5. We qualitatively evaluate feasibility of guided configuration evolution.
6. We quantitatively evaluate guidance by analyzing evolution of real-world product lines and existing configurations.

This work is an extended and revised version of prior work [23]. Contributions 5–6 are novel with respect to prior work. In addition, we add illustrations and descriptions regarding the general idea of guided configuration evolution as well as the process for applying templates. Furthermore, we extend the explanation on how to apply our method by incorporating tools that classify product line changes, which can be used to automatically apply or identify matching templates.

2 Behavior preservation

Given a configuration, the respective product generated before and after product-line evolution may behave differently due to changes to artifacts that are mapped to the selected features. In the following, we define our notion of product behavior. To this end, we first introduce basic product-line concepts by means of exemplary car product line. The respective product-line artifacts, i.e., the feature model, the reusable implementation artifacts, and the feature-artifact mapping, are depicted in Fig. 1. We adapt existing notions and formalisms for product lines [22,24]. We formalize a feature set F as the set of all features. For instance, in our example, this is the `Car` root feature, two features `AudioCD` and `Cassette` for entertainment functionality, two features `GPS` and `Glonass` responsible for different positioning services, and a feature `Engine` with two alternative manifestations in terms of the features `Diesel` and `Gasoline`. We abstract from feature relations or other constraints, as our notion of product behavior is independent of such relations and we do not tie our concepts to a particular feature model representation. 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 feature `GPS` can be realized using a plug-in `car.gps`. In

[1] https://gitlab.com/DarwinSPL/GuyDance.

[2] https://gitlab.com/mnieke/guydance_proofs.

a *feature-artifact mapping*, features are related to reusable artifacts [5]. For instance, in the running example, a feature-artifact mapping with preprocessor directives could look like: #if Engine <code> #endif. We abstract from concrete implementation and feature-artifact mapping techniques, such as programming languages or concepts. We consider a feature-artifact 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 contained in $\mathcal{P}(I)$. For instance, a feature-artifact mapping entry could look like: $M(\texttt{GPS} \vee \texttt{Glonass}) = \{\texttt{car.positioning}\}$.

We consider a product line as a triple $PL = (F, I, M)$ with the feature set F, the set of reusable artifacts I, and the feature-artifact mapping M. We assume that the artifacts in M are present in I. Our methodology is deliberately oblivious of configuration validity as its goal is to automate configuration updating as far as possible with a focus on product behavior. Thus, configuration validity is out-of-scope and we do not consider the constraints between features. A product can be generated by composing all reusable artifacts that are collected using the mapping and a configuration. In particular, 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 synonymously, we adopt the distinction between those two elements from the literature and consider a configuration as an implementation-agnostic set of features, whereas a product comprises the implementation generated for a configuration [25].

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

To describe evolution of feature-artifact mapping, we define a replace operator $M[exp_0 \mapsto exp_1]$ that iterates over all application conditions of M and replaces occurrences of the feature expression exp_0 by the feature expression exp_1. In the running example, if Diesel should be replaced by Engine in the feature-artifact 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 $i' \notin I, i' \in I'$. This makes our notation independent of concrete artifact language and artifact evolution operation, as, if an artifact i was changed, we consider it to be a new different artifact i' after evolution.

We formalize product behavior and its preservation. As, in general, program behavior equality is undecidable [26], we rely on a more conservative notion for comparison. As approximation for product behavior, we use the definition of

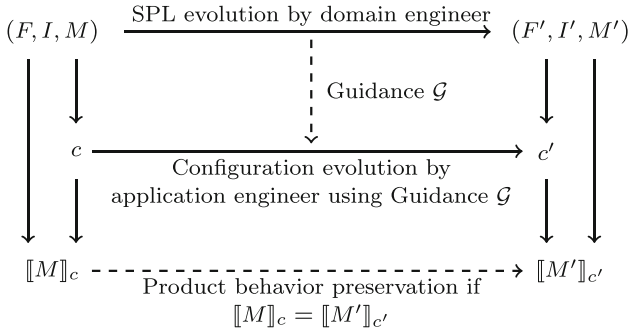


Fig. 2 Commuting diagram of updating a configuration after product-line evolution

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 exploit the fact that syntactic equality preserves product behavior.

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 GPS is mapped to $i_{\texttt{Glonass}}$. 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 (e.g., $c' = c \setminus \{Glonass\}$ in the example above). We identify configuration subsets that need to be updated by adapting the filter operator $\restriction$ of Sampaio et al. [22]. For a feature set F and a feature expression exp, $F \restriction exp$ yields the set of configurations of F that satisfy exp. For instance, in the running example, if AudioCD and Cassette are deleted, all configurations that select any of these features need to be updated, i.e., the configurations yielded by $F \restriction \texttt{AudioCD} \vee \texttt{Cassette}$.

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 was deleted. We provide a methodology to support domain engineers in guiding application engineers on updating their configurations.

Table 1 Guidance for a *Delete Feature* operation

	Operation: delete feature f_0 with realization artifacts (r)				
	Configuration subsets	Update operations	Preserves behavior	Update Rationale	Type
(x_1)	$Delete_0 : c \in F \upharpoonright \neg f_0$ (S_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)	$Delete_1 : c \in F \upharpoonright f_0$ (S_2)	$c' = c \setminus \{f_0\}$ $(op_{2,1})$	No $(b_{2,1})$	f_0 Not available anymore. $(r_{2,1})$	Semi-autom(t_2)

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 which of the suggested configuration update operations to perform to find a configuration that best suits their use case. Application engineers can use guidance at a time of their choice and independently of domain engineers to update configurations that are relevant to them. Figure 2 shows the general idea of our contribution. For a product line $PL = (F, I, M)$, 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 configuration to c' and corresponding product $[\![M']\!]_{c'}$, which can be derived from PL'. Depending on the intent of the evolution operation, the defined guidance may preserve product behavior. However, in all cases, domain engineers have to specify whether product behavior is preserved by applying the provided guidance. In particular, we conservatively assume that product behavior is not preserved if resulting products use different implementation artifacts than before evolution (cf. Sect. 2).

3.1 Structure of configuration evolution guidance

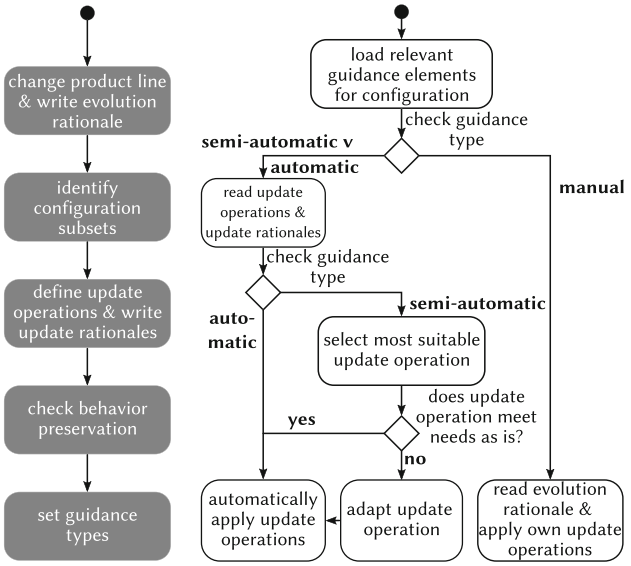
Configuration evolution guidance consists of the essence of product-line evolution, configuration update suggestions, and statements of product behavior preservation. Table 1 shows an example of guidance for a *Delete Feature* evolution operation. For easier reference, we added identifiers in brackets 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 were performed. In the example of Table 1, it is stated that the feature f_0 is deleted together with all artifacts that are mapped to it.

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 a large subset of configurations. In the example, the guidance element x_1 contains S_1 which defines the subset of all configurations *not* selecting the deleted feature. The guidance element x_2 contains S_2 which defines the subset of configurations that select f_0.

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 ($r_{i,j}$) that explains why they defined this operation and in which cases it is sensible to apply the update operation. In the example, U_1 (only constituent elements shown in Table 1) is the set of possible configuration update operations for the guidance element x_1. It contains one update operation $u_{1,1}$ that has one set operation $op_{1,1}$. This operation establishes that configurations should be left unchanged. The rationale $r_{1,1}$ of this update operation states that these configurations can remain unchanged as they are not affected by the evolution.

Additionally, domain engineers specify whether product behavior is preserved ($b_{i,j}$) by applying an update operation (i.e., $u_{i,j} \in \mathcal{U}_i = (op_{i,j}, r_{i,j}, b_{i,j})$). In this way, application engineers know whether they can expect stable behavior after evolution, or whether they need to perform additional work. For instance, if product behavior did change, the product has to be tested again in order to avoid error during runtime. In the example, it is specified that the update operation $u_{1,1}$ will preserve product behavior ($b_{1,1}$). The second guidance element x_2 is specified analogously to x_1 for configurations selecting the deleted feature. However, this feature must be removed from the respective configurations ($op_{2,1}$) as the feature does not exist anymore ($r_{2,1}$) which results in changed product behavior ($b_{2,1}$).

Guidance can be applied on different levels of automation. The type of a guidance element (t_i in Table 1) indicates this automation degree and can be *automatic*, *semi-automatic*, or *manual*. Guidance with type automatic indicates that the respective configuration update operation can be applied without any manual effort from application engineers. Semi-automatic guidance indicates that manual effort is required



(a) For domain engineers.

(b) For application engineers.

Fig. 3 Guided configuration evolution process. **a** For domain engineers. **b** For application engineers.

from application engineers in terms of choosing between multiple possible configuration update operations that can then be applied automatically. Guidance has manual type if no configuration update operation is specified and application engineers need to specify update operations on their own using the information on the product-line evolution. Manual guidance is required if domain engineers are not able to define update operations. Guidance $\mathcal{G} = (r, \mathcal{X})$ is defined as a tuple containing the evolution rationale and the set of guidance elements. For instance, the guidance of the example of Table 1 is described for an evolution operation that deletes feature f_0. Two guidance elements (x_1, x_2) are defined for configuration subsets not selecting f_0 (S_1) and selecting f_0 (S_2). Configurations that do not select f_0 can remain as-is $(op_{1,1})$ which preserves behavior $(b_{1,1})$ as the configurations are not affected by the evolution $(r_{1,1})$, and this configuration update can be applied automatically (t_1).

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 3a 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. Optimally directly afterward or even before, so that no details on the evolution are forgotten, domain engineers define guidance. To share knowledge on 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, domain engineers 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 Fig. 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, domain engineers define one or multiple update operations, and rationales explaining them with their impact on configurations. Multiple update operations are necessary if the domain engineer identifies several sensible possibilities to update certain 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 guidance with manual type 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 feature-artifact mapping before and after evolution. Different levels of product behavior assurance may be defined by domain engineers. In the following, we give three exemplary levels that may serve as a sensible basic level set, but domain engineers can define other levels as well in accordance with the domain or scenario. 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 without alternatives for update operations, 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 be used only 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, domain engineers set the type to *semi-automatic*. We consider it as undisciplined usage if no update operation was defined, and set the type to *manual*.

Figure 3b 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. First, application engineers evaluate 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 upfront effort. To reduce effort for domain engineers, we provide the possibility to store guidance for evolution scenarios in the form of templates to facilitate reuse. Consequently, guidance templates further automate the presented process, but are not necessary to apply our method, e.g., for update operations that are used only once. In contrast to guidance without templates, 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 feature-artifact mapping evolution (e_M), described in terms of the evolution operations we defined in Sect. 2. The evolution operations are preconditions for applying the guidance defined in the templates. Thus, an evolution template $\mathcal{T} = (\mathcal{E}, \mathcal{G})$ 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 [22,27–29]. The templates also illustrate the general concept of guided configuration evolution. To better reference the table's elements in the text, we add identifiers for guidance elements and update operations.

4.1 Delete feature

It is expensive to maintain a large set of features and their realization artifacts so that we define the *Delete Feature* template to remove obsolete features from the product line. Maintaining certain features may not be profitable anymore. In the running example (cf. Fig. 1), cassettes are rarely used and almost no customer selects the feature. Therefore, this feature is deleted, including its mapped artifacts.

We used the guidance of this template to illustrate the structure of guided configuration evolution in Sect. 3.1. Table 2 shows the elements of this template, and thus, mostly resembles Table 1. Therefore, we only describe the product-line evolution which is the precondition for applying this template. As precondition, the feature f_0 is removed from the feature set and in the feature-artifact mapping, it is replaced by *false*. The first guidance element ($Delete_0$) addresses the configuration subset not selecting f_0. We specify the type as *automatic* because such configurations remain unchanged, as they are unaffected by the operation, and explain this in the rationale.

4.2 Merge features

When systems evolve, individual features may grow together into one semantic unit [27] for which we define the *Merge Features* template. In our running example, new cheaper hardware is capable of providing functionality for both features `GPS` and `Glonass`. Thus, `Glonass` is merged into `GPS`.

Table 3 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 the feature-artifact mapping.

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 type 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 type to *automatic*.

The third guidance element $Merge_2$ is for configurations containing f_0 but not f_1. As f_0 still exists and configurations that select this feature might still be valid after evolution, existing approaches [16,20,21] detecting defects in configurations would leave the configuration as-is. However, as f_0 also provides the functionality of f_1 after evolution, with our methodology, we know that the respective products *do not* preserve behavior. Without this knowledge, products with syntactically similar configuration but with altered behavior might be deployed, which may cause harm. Thus, 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. For application engineers who do not want to have the additional functionality of f_1 in the products, we provide a second update operation $M_{2.b}$ that removes

Table 2 Template *Delete Feature*

Operation: delete feature f_0 with realization artifacts
$F' = F \setminus \{f_0\}$, $M' = M[f_0 \mapsto false]$

Configuration subsets	Update operations	Preserves behavior	Update rationale	Type
$Delete_0 : c \in F \restriction \neg f_0$	$c' = c$	Yes	Not affected. Can be left as-is	Autom
$Delete_1 : c \in F \restriction f_0$	$c' = c \setminus \{f_0\}$	No	f_0 Not available anymore	Semi-autom

f_0 from configurations. As domain engineers 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, i.e., for configurations that do not select f_0 but select f_1 and is defined similarly to $Merge_2$. As f_1 does not exist anymore after evolution, it must be removed from all configurations selecting it. We define two configuration update operations for this case. $M_{3.a}$ removes f_1 from the respective configurations. As this results in loss of f_1's functionality, we define that product behavior is not preserved. In the second configuration update operation $M_{3.b}$, f_1 is removed as well, but f_0 is added. As a result, products of respective configurations provide the functionality of both, f_0 and f_1, after evolution, similar to $M_{2.a}$. Consequently, product behavior is not preserved as it additionally contains the original functionality of f_0. Again, we set the guidance type to *semi-automatic* as product behavior cannot be preserved and application engineers have to decide whether they want to lose the functionality of f_1 or can accept the additional functionality of f_0.

4.3 Extract new feature

To allow more precise configuration, parts of a feature's functionality can be extracted into a separate feature so that we define the *Extract New 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`. The *Extract New Feature* template shifts functionality from a *source* feature into a new *target* feature.

Table 4 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 feature-artifact mapping. We identified four cases: first, if an artifact remains mapped to f_0 after evolution, we leave the feature-artifact 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 $E_{0.a}$ leaves corresponding configurations unchanged and preserves product behavior. The update operations $E_{0.b}$ and $E_{0.c}$ add f_0 or f_1, respectively, to the configuration. 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 $E_{1.a}$ adds the feature f_1 to the configurations as described above. The second operation $E_{1.b}$ leaves the configuration as-is. The resulting product's functionality is reduced by the extracted functionality of f_1. The third operation $E_{1.c}$ 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 [16,20,21] 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.

Table 3 Template *Merge features*

Operation: merge functionality of feature f_1 into feature f_0
$F' = F \setminus \{f_1\}$, $M' = M[f_1 \mapsto f_0]$

Configuration subsets	Update operations	Preserves behavior	Update rationale	Type
$Merge_0 : c \in F \restriction (\neg f_0 \wedge \neg f_1)$	$c' = c$	Yes	Not affected. Can be left as-is	Autom
$Merge_1 : c \in F \restriction (f_0 \wedge f_1)$	$c' = c \setminus \{f_1\}$	Yes	Functionality of both selected features f_0 and f_1 are now covered by f_0. f_1 does not exist anymore	Autom
$Merge_2 : c \in F \restriction (f_0 \wedge \neg f_1)$	$M_{2.a} : c' = c$	No	Leave as-is if the additional functionality of feature f_1 is acceptable	Semi-autom
	$M_{2.b} : c' = c \setminus \{f_0\}$		Remove feature f_0 if additional functionality of feature f_1 is not acceptable. Results in loss of f_0's functionality	
$Merge_3 : c \in F \restriction (\neg f_0 \wedge f_1)$	$M_{3.a} : c' = c \setminus \{f_1\}$	No	Remove feature f_1 if additional functionality of feature f_0 is not acceptable. Results in loss of f_1's functionality	Semi-autom
	$M_{3.b} : c' = (c \cup \{f_0\}) \setminus \{f_1\}$		Replace feature f_1 by f_0 if the additional functionality of feature f_0 is acceptable	

Table 4 Template *Extract New Feature*

Operation: extract some functionality of feature f_0 into a new feature f_1
$F' = F \cup \{f_1\}, \mathcal{M}' \subseteq \{m' = m, m' = m[f_0 \mapsto f_1], m' = m[f_0 \mapsto (f_0 \wedge f_1)], m' = m[f_0 \mapsto (f_0 \vee f_1)] \mid m \in \mathcal{M}\}$

Configuration subsets	Update operations	Preserves behavior	Update rationale	Type
$Extract_0 : c \in F \upharpoonright \neg f_0$	$E_{0.a} : c' = c$	Yes	Not affected. Can be left as-is	Semi-autom
	$E_{0.b} : c' = c \cup \{f_0\}$	No	If the functionality of f_0 without f_1 is desired, f_0 can now be added	
	$E_{0.c} : c' = c \cup \{f_1\}$		If the functionality of f_1 without f_0 is desired, f_1 can now be added	
$Extract_1 : c \in F \upharpoonright f_0$	$E_{1.a} : c' = c \cup \{f_1\}$	Yes	To preserve product behavior, add f_1	Semi-autom
	$E_{1.b} : c' = c$	no	If the functionality of f_0 without f_1 is desired, it can be left as-is	
	$E_{1.c} : c' = c \setminus \{f_0\} \cup \{f_1\}$		If the functionality of f_1 without f_0 is desired, replace f_0 by f_1	

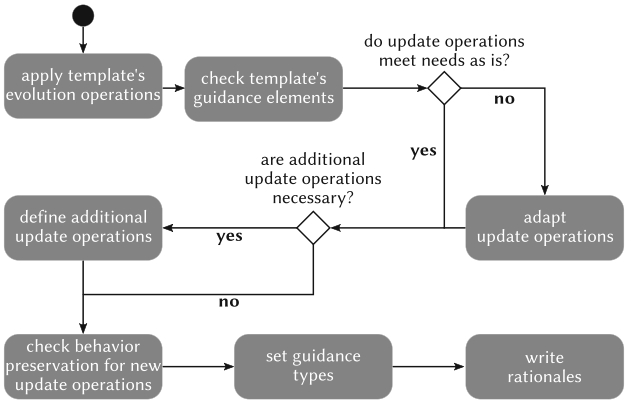


Fig. 4 Guided configuration evolution process for domain engineers using templates

4.4 Evolution process with templates

The presented templates are examples that illustrate the usage of guided configuration evolution, and we do not claim completeness of the template set. Hence, as additional templates may be necessary, we enable domain engineers to define their own templates. However, since it is not the case that each guidance might be worth a template (e.g., one-off occurrence of an evolution operation), our methodology can also be applied without templates following the process defined in Sect. 3.2.

To cover the guided configuration evolution with and without templates, we need to adapt the process defined in Sect. 3.2. Figure 4 shows the process for applying a template from domain engineers' perspective. After selecting a template to be used, domain engineers perform the defined evolution operations for the feature set and the feature-artifact mapping using the tools they are used to. Using suitable tooling, this step can be further automated. After performing evolution operations, a tool can help in identifying suitable templates from a template catalog. As the feature set and feature-artifact 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 no additional or modified update operations are necessary. However, domain engineers should always check whether they can define additional domain-specific update operations to better guide application engineers for a concrete evolution scenario. If the update operations are not completely matching the evolution scenario or the domain engineers' requirements, existing update operations can be adapted or can be 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 specify that a new update operation preserves product behavior, they have to ensure that the resulting artifact set is the same after the update, e.g., with a formal proof or excessive testing. In the next step, the guidance types of the guidance elements must be set. We deliberately define this as a mandatory step to stimulate domain engineers in providing more information. We expect that as much domain-specific information as possible helps application engineers in fixing their configuration. Finally, the rationales for the evolution operation and the update operations have to be written. This is of particular importance as application engineers 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, it can be fully automated resulting in no manual effort. If no existing template is suitable or an existing template must be adapted, the defined guidance can be saved as new template. These new 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 product-line evolution is formally defined in the templates, this serves as basis for automatically selecting suitable templates when evolution scenarios can be detected from performed changes. 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 (e.g., if an evolution operation is too specific to save it for reuse), it can be gradually extended by templates, existing templates can be reused directly, or existing templates can be adapted for a concrete scenario.

5 Proving behavior preservation

We fully formalized proofs for behavior preservation of the three templates using the theorem prover *PVS* [30]. For the sake of brevity, we provide sketches of those proofs in the following and provide the complete proofs via our online repository.[3]

[3] https://gitlab.com/mnieke/guided-config-evo-eval-data.

We utilize the formalization that we introduced in Sect. 2, i.e., product behavior of a configuration c is preserved by c', if $[\![M]\!]_c = [\![M']\!]_{c'}$. For the *Delete Feature* template, behavior is preserved for configurations that did not select the deleted feature f_0 (cf. Table 2, $Delete_0$). 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* $M' = M[f \mapsto false]$*:*

$$\forall c \in F \upharpoonright (\neg f) : [\![M]\!]_c = [\![M']\!]_c$$

The idea of the proof is that we can show that 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 all feature-artifact mapping entries. 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 3, $Merge_0$ 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:

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

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

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 4, $Extract_{0.a}$ and $Extract_{1.a}$). 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* $\mathcal{M}' \subseteq \{m' = m, m' = m[f_0 \mapsto f_1], m' = m[f_0 \mapsto (f_0 \wedge f_1)], m' = m[f_0 \mapsto (f_0 \vee f_1)] \mid m \in \mathcal{M}\}$*:*

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

Successful application of our method requires that the templates, and especially, the feature set and feature-artifact mapping evolution operations are applied correctly. To ensure this correct application, tool support is beneficial that either applies the evolution operations automatically or verifies whether performed evolution matches the pre-conditions of a template.

6 Applying guided configuration evolution

Tool support is pivotal for practitioners to use 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 Figs. 3a, b and 4. We implemented an early open-source prototype, named *GuyDance*,[4] to show feasibility.

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 minor, our method does not have to be applied, but it can be applied for other changes or even retroactively when first problems occur.

To define guidance, 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. For this last option, the level of automation can be increased with additional tool support. As highlighted in Sect. 4.4, template evolution operations on the feature set and the feature-artifact mapping can be applied automatically and, subsequently, the guidance of that template can be reused. Another option is to automatically identify templates that match the evolution of the feature set and the feature-artifact mapping. For instance, the tool FEVER [31] is able to extract and detect changes that match a certain pattern, such as evolution scenarios described in templates. To define or adapt guidance and respective templates, a domain-specific language that provides the possibility to specify respective information is most suitable. In *GuyDance*, we used Xtext[5] for defining a grammar and editors for guidance and 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 or a simple Boolean evaluation algorithm. For instance, if a subset is defined as $c \in F \upharpoonright \neg f_0$ and a configuration selects features f_1, f_2, the SAT solver 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, based on the guidance type, configuration update operations are (semi-)

[4] https://gitlab.com/DarwinSPL/GuyDance.

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

automatically applied or have to be manually performed by the application engineer. 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 an update operation, selected features of an existing configuration are either deselected or newly selected features are added to that configuration.

A characteristic of our methodology is that it does not depend on an input configuration being valid. It is capable of using an invalid configuration as input to apply update operations. As output, we do not limit the approach to valid configurations, but deliberately decided to allow invalid configurations as output. Thus, it is not required to check configuration validity after each evolution step. In contrast, we allow to perform multiple/all required update operations first and then check for validity, e.g., using interactive configuration editors [1,20,32]. Otherwise, configuration validity would have to be reestablished after each evolution step. As new product-line versions typically contain multiple evolution steps and configurations are not necessarily updated to each new product-line version but skip multiple versions, updating to a new product-line version would lead to many validity checks and modifications to configurations. Additionally, if a configuration becomes invalid in an intermediate evolution step but would remain valid after the application of all evolution steps, this would lead to unnecessary divergences from the original configuration. Apart from that, if configuration validity has been restored first, finding a configuration that preserves product behavior may render this configuration invalid and, thus, requiring to restore configuration validity again. We expect that most configurations remain valid if product behavior can be preserved. If this is not the case, it is most likely that no valid configuration exists that preserves product behavior as a different but valid configuration would result in different product behavior. As a result, we propose to consider configuration validity only after applying the guidance of all product-line evolution steps that have been performed since the last configuration update. With our methodology, we take the first step in the process of updating configurations to new product-line versions. Our goal is to achieve a high degree of automation with a focus on product behavior. It remains an open research question whether it is better to reestablish configuration validity after each evolution step, at the end of multiple evolution steps, or in combination with product behavior.

7 Evaluation

Our goal with guided configuration evolution is to *automate the process of updating configurations semantically as far as possible*. To show that we achieve this goal, we perform three complementary evaluations: First, we have proven behavior preservation in *PVS* for typical evolution scenarios captured by our pre-defined templates (cf. Sect. 5). Second, we perform a qualitative evaluation of the guidance templates and methodology based on the real-world evolution of our industry partner's product line (cf. Sect. 7.1). Third, we perform a quantitative evaluation based on the real-world evolution of the Linux kernel product line and based on real-world configurations of two product lines (cf. Sect. 7.2). With the latter two evaluations, we seek to answer the following main research question:

> RQ_{main} Is it feasible to apply guided configuration evolution to real-world product-line evolution?

We pose individual research questions for each evaluation which are defined in the respective subsections.

7.1 Qualitative evaluation

In the qualitative evaluation, we investigate the following research questions:

> *RQ1a* Is it feasible to adapt guidance templates to fit real-world product-line evolution?
> *RQ1b* Is it feasible to derive new guidance templates from real-world product-line evolution?
> *RQ1c* What is the effort to define guidance for real-world product-line evolution?

To retrieve realistic data for product-line evolution and intended configuration update operations, we interviewed our industry partner *Schnapptack* (https://schnapptack.de/) about the evolution of their web application product line and problems that emerged during this process. With their Software Product Line (SPL), they can derive custom-tailored web applications that are based on different visual blocks of which the web application can be composed. The project is medium scale with around ten developers working on it. We identified ten evolution scenarios by interviewing the project leader on the product-line evolution. Then, we applied our pre-defined templates if possible. If the three example templates were not sufficient and, thus, could not be applied as-is, we adapted these templates to fit the scenarios to answer RQ1a. If no existing template is suitable for a scenario, we define new guidance and, if possible, derive a new template from that scenario to answer RQ1b. Finally, we investigated how much effort was spent to apply and adapt the templates to answer RQ1c.

Figure 5 shows an excerpt of the feature model provided by *Schnapptack* before evolution. Most of the features represent blocks, which are visual components to build web applications. The original feature model contains 111 features but

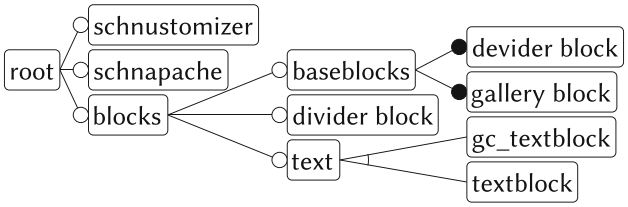


Fig. 5 Excerpt of Schnapptack's feature model

we omit parts not affected by evolution. In the following, we explain five scenarios we identified in the interview. The remaining five evolution scenarios were similar to the following ones, which results in the same insights.

7.1.1 Scenario 1

The feature `baseblocks` groups all basic block features (e.g., `devider block`, `gallery block`, …) and the data model of all basic blocks was mapped to `baseblocks`. To allow more fine-grained configuration options, all basic blocks become `optional` and the data models for each basic block feature are extracted from `baseblocks` to the respective features. Finally, `baseblocks` is deleted. We adapted the *extract feature* template to map artifacts to an existing feature. We performed this scenario for each of the sub features. Then feature `baseblocks` is removed using the *delete feature* template.

7.1.2 Scenario 2

By mistake, two features implement the same functionality (`deviderblock` and `dividerblock`). Therefore, `deviderblock` is removed together with its mapped artifacts. We captured this scenario by applying the *delete feature with mapped artifacts* operation. As we know that both features implement the same functionality, we adapted the template so that it replaces `deviderblock` with `dividerblock` in configurations. This does not preserve product behavior as different artifacts are used, but we explained in the rationale that product behavior is similar. However, without our method, a configuration update operation would just deselect the feature `deviderblock` in configurations which would unexpectedly result in changed product behavior.

7.1.3 Scenario 3

The features `gc_textblock` and `textblock` implement similar functionality and share code mapped to `text`. The `textblock` is the more mature feature, but the `gc_textblock` feature implements additional bug fixes for the shared functionality. Thus, both features are merged into `text` and we applied the *merge feature* template twice.

7.1.4 Scenario 4

A library of the feature `schnustomizer` is used by other features. Some features have a dependency to `schnustomizer` only to use this library. Other features are mapped to copies of this library (even in old versions). To resolve the unnecessary dependency to other functionality of `schnustomizer` and to resolve the redundant library copies, we created a new feature and map the library to it instead of `schnustomizer`. Additionally, we removed the feature-artifact mapping of all other features to the library. We captured this scenario using the *extract new feature* template on feature `schnustomizer`. As an original configuration that selects `schnustomizer` might still be valid, product behavior would change, and without configuration guidance, application engineers might be unaware. For configurations that select other features containing copies of the library before evolution, we defined additional configuration update operations that are equal to $E_{0.c}$ and set the guidance category to automatic. Thus, the new feature is automatically selected in respective configurations without the need for interaction from application engineers.

7.1.5 Scenario 5

In the last scenario, a feature `nginx` is introduced to supersede feature `schnapache`. The feature `nginx` is now the default webserver, but the `schnapache` webserver is still available. For this scenario, we created a new template containing two configuration update operations: First, to replace `schnapache` with `nginx` in all configurations and, second, to leave each configuration as-is. As `schnapache` is still valid, we set the guidance category to *semi-automatic* and write in the rationale that we encourage using `nginx` but that it is not compulsory. The first configuration update operation does not preserve product behavior, but the second one does.

7.1.6 Discussion

We were able to capture all evolution scenarios of *Schnapptack* and provide sensible configuration update operations. In more details, for Scenarios 1–4, we reused our pre-defined templates. For Scenarios 1, 2, 4, we had to adapt the template configuration update operations to fit the scenarios. For Scenario 2 in particular, we were able to simulate a *replace feature* operation by adapting the *delete feature* template, which shows the flexibility of our method. Thus, we are able to adapt guidance templates to fit real-world product-line evolution (RQ1a).

In Scenario 5, we could not reuse any template but had to define completely new guidance. This required to write the rationale, to identify the relevant configuration subsets (i.e., $c \in F \restriction \mathit{Schnapache}$ and $c \in F \restriction \neg \mathit{Schnapache}$), and to define configuration update operations for those configurations. In particular, we defined two possible update operations $c' = c$ and $c' = (c \setminus \{\mathit{Schnapache}\}) \cup \{\mathit{nginx}\}$ for configurations that select *Schnapache*. Configurations that do not select *Schnapache* can remain as-is, i.e., $c' = c$. We defined a *replace feature* template based on this scenario, and thus, we are able to derive new guidance templates from real-world product line evolution (RQ1b).

Adapting existing (RQ1a) and deriving new templates (RQ1b) requires deep knowledge of the variability of the considered product line and foresight regarding the impact of changes on configurations. However, we argue that defining and adapting templates is in the same problem domain as defining and modifying feature models or variable implementations, e.g., #ifdef annotations. Thus, domain engineers who are responsible for feature models and variable implementations should be able to define and adapt guidance. Moreover, with an increasing catalog of guidance templates, we envision that most of the occurring evolution operations are covered by an existing template. Therefore, less experienced domain engineers are able to reuse these templates which makes the entire evolution process even easier for them than without our method. For the remaining cases which are not covered by existing templates, experts who are experienced in defining guidance can be consulted.

The effort we had to spend to define guidance (RQ1c) slightly differs for each scenario. In Scenario 1, we adapted the existing *extract feature* template and instead of extracting functionality to a new feature, we used an existing feature as target. In particular, the only change to the template was to remove the feature-set evolution operation. Then, we applied the *delete feature* template which resulted in almost no additional costs as it already existed and we just had to select the feature to be deleted and apply the evolution operation.

For Scenario 2, we adapted the *delete feature* template to replace the deleted feature in configurations by another feature. Thus, we only had to modify the template configuration update operation correspondingly ($c' = (c \setminus \{\mathit{deviderblock}\}) \cup \{\mathit{diviederblock}\}$).

In Scenario 3, the templates could be applied as-is and, thus, this resulted in no additional effort. In Scenario 4, we had to adapt the *extract new feature* template. The main challenge was to identify the features with library copies which was done together with our interview partner. In Scenario 5, we defined completely new guidance and a new template as described above.

In contrast to real-world application of our methodology, we as external researchers defined the guidance for the scenarios. To this end, we interviewed our industry partner to understand the product line and what is planned for the next evolution steps. To better understand the necessary effort, we give some quantitative data in terms of time we spent to define the guidance for the scenarios. In particular, the interview was split in two sessions and took about 2 hours. However, this also included understanding the product line and the feature model, which would not be necessary for engineers who are familiar with that product line. In summary, we spent about 20 minutes per scenario. Most effort was spent for identifying relevant features in Scenario 4 and for defining the entire guidance for Scenario 5. However, the latter took us about ten minutes which is little time compared to investigating all the company's configurations individually. Additionally, by modeling guidance with our approach, we persisted knowledge on how to update configurations that can be used at a later point in time—even when domain engineers already forgot what they exactly changed and why.

7.2 Quantitative evaluation

In the quantitative evaluation, we investigate to which extent the guided configuration evolution is helpful for large-scale product lines. In particular, we are interested in the provided automation degree and the number of configurations for which we provide additional benefit compared to existing methods. This results in the following research questions:

RQ2a For which percentage of configurations can we provide full automatic guidance?
RQ2b For which percentage of configurations is knowledge of both domain engineers and application engineers required?
RQ2c For which percentage of configurations can we ensure behavior preservation?
RQ2d For which percentage of configurations do we detect product behavior changes that would not be detected using existing methods?

To answer these questions, we use hundreds of evolution operations and investigate the impact on thousands of configurations. In particular, we consider evolution operations that match the templates we defined in Sect. 4. As previous work identified these evolution operations as relevant [22,27–29], we use them as representative subset of possible evolution operations. For each occurrence of an evolution operation, we investigate, for each configuration, which guidance element is applicable. In particular, we identify the relevant configuration subset of the guidance elements (rows in Tables 2, 3, 4) that covers the considered configuration.

We perform two complementary quantitative evaluations with three real-world product lines that correspond to the different roles for the evolution of product lines and configurations: The first quantitative evaluation represents the

domain engineer's perspective who performs and knows about the actual product-line evolution but does not know existing configurations. To this end, we use the real-world evolution. The second quantitative evaluation represents the application engineer's perspective who knows about the maintained configurations but does not perform product-line evolution. For this evaluation, we use their real-world configurations. For both evaluations, from domain and application engineer's perspectives, we measure which configurations are covered by which guidance element as described above. The evaluation software and all data can be found in our online repository (cf. Footnote 3).

7.2.1 Setup of domain engineer's perspective

For the evaluation from domain engineer's perspective, we analyze the Linux kernel evolution. From its development history, we search for commits that correspond to the templates of Sect. 4 using the tool *FEVER* [31] and extract the feature model using the tool *KConfigReader* [33]. We use the first six commits found by *FEVER* corresponding to each template. For this purpose, we analyze commits between versions 2.6.28 and 3.16. The analyzed feature model has between **8003** (version 2.6.34) and **16,542** (version 3.16) features, depending on the kernel version and analyzed architecture (i.e., x86, ARM, MIPS, or PowerPC).

Ideally, we would generate all possible configurations to analyze the evolution's impact on them. However, it is not feasible to enumerate all configurations due to exponential growth of the number of configurations relative to the features [34]. For domain engineers, configurations appear to be random as they do not know which configurations are used and why. Thus, to retrieve realistic results, we randomly generate configurations using *FeatureIDE* v3.3 [35]. For each commit, we generate 1,000 valid configurations.

7.2.2 Setup of application engineer's perspective

For the evaluation of the application engineer's perspective, we use two real-world product lines and their real-world configurations [36–38]. Product line *Agrib* consists of **2008** features and **5749** configurations. Product line *ERP* consists of **1728** features and **170** configurations. As we did not have access to multiple versions of both product lines and evolution appears random to application engineers, we randomly generate evolution scenarios for both product lines based on the evolution operations of the predefined templates (cf. Sect. 4). We generate 100 random operations for each template, resulting in 300 evolution operations for each product line. Although we randomly generate the evolution scenarios, we evaluate all available real-world configurations for each scenario.

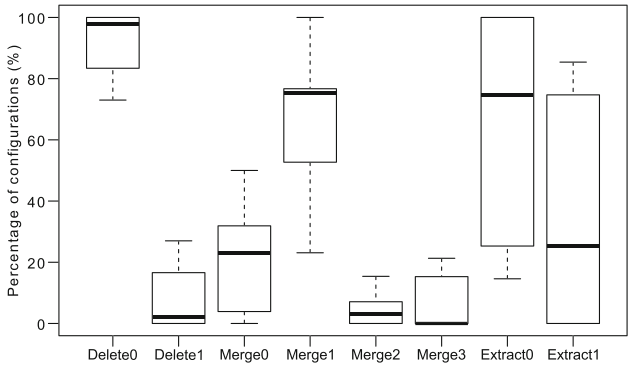


Fig. 6 Guidance for real-world evolution of the Linux Kernel with six evolution operations for each template and 1000 configurations (domain engineer's perspective)

7.2.3 Results of the quantitative evaluation

For both quantitative evaluations, we analyze, for each evolution operation, how many configurations are covered by which guidance element of the templates. Therefore, we are able to determine for each configuration and evolution operation whether application engineers can automatically apply guidance and whether product behavior can be preserved. For instance, if a configuration is covered by the guidance element $Merge_0$ after applying a merge features evolution operation, the configuration update operation can be automatically applied and product behavior is preserved (cf. Table 3).

Figures 6 (domain engineer's perspective) and 7 (application engineer's perspective) show the aggregated results of the quantitative evaluation. Each data point represents one evolution operation and shows the percentage of configurations covered by the respective guidance element. For instance, if 100 configurations exist, the *delete feature* template is applied and 80 configurations are covered by $Delete_0$, a data point at 80% for $Delete_0$ is added. For simplicity and as the results are very similar, we use average values of the evaluation of the both product lines, *Agrib* and *ERP*, to describe the results for the application engineer's perspective.

The results for both perspectives show similar patterns. For *delete feature* evolution operations, most configurations are covered by the guidance element $Delete_0$ (cf. Table 2), i.e., the deleted feature is not contained by the respective configurations. In the median, 97.9% of the configurations for the domain engineer's perspective (Linux kernel, cf. Fig. 6) and 95.2% of the configurations for the application engineer's perspective (*Agrib* and *ERP*, cf. Fig. 7) are covered by $Delete_0$. This is most likely the case as core features that are part of many configurations, i.e., which many other features are dependent on, are typically not deleted in real-world evolution. It is more likely that such features are successively cut off from other features, and, finally, deleted when no other

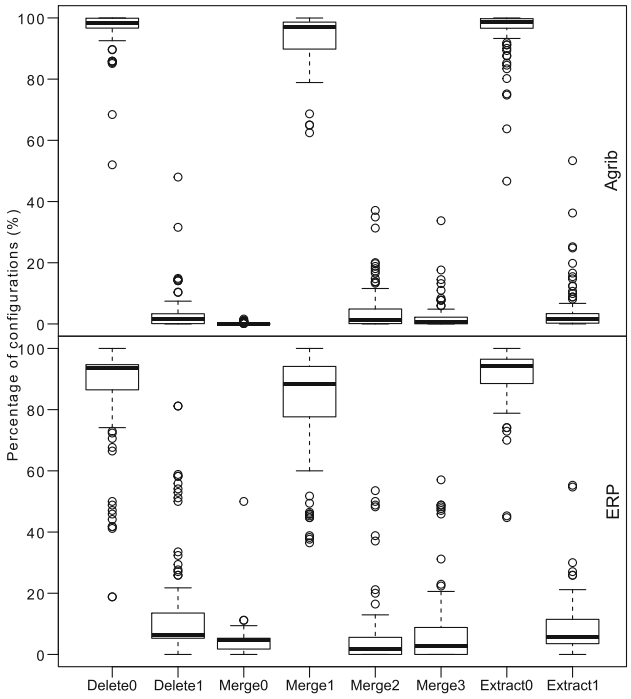


Fig. 7 Guidance for real-world configurations for 100 applications of each template evolution operation for the product lines *Agrib* and *ERP* (application engineer's perspective)

features depend on them. Additionally, the results from application engineer's perspective indicate that only few of such core features exist at all.

Regarding the *merge features* evolution operations, most configurations are covered by the guidance element $Merge_1$, i.e., these configurations contain both merged features. The respective median for the Linux kernel is 75.2% (domain engineer's perspective, cf. Fig. 6) and for *Agrib* and *ERP*, it is 92.9% (application engineer's perspective, cf. Fig. 7). The second most configurations for the domain engineer's perspective are covered by the guidance element $Merge_0$ (in the median 23.0%), i.e., these configurations contain none of the merged features. This shows that those features that are merged are typically selected in combination or not at all.

For the *extract new feature* template, the results from domain engineer's perspective have a similar trend as the results from application engineer's perspective, but differ in the degree of scattering. In general, most configurations are covered by the guidance element $Extract_0$ (the source feature is not selected). For the domain engineer's perspective (Linux kernel, cf. Fig. 6), the median value is at 74.8%, whereas for the application engineer's perspective (*Agrib* and *ERP*, cf. Fig. 7), it is at 99.5%. The lower quartile of the configurations covered by the guidance element $Extract_0$ is at 31.9% for the Linux kernel (cf. Fig. 6), whereas it is at 96.6% for *Agrib* (cf. Fig. 7 top) and at 88.7% for *ERP* (cf. Fig. 7 bottom). In summary, this means that it is rather improbable that functionality is extracted from an important feature that is part of many configurations. However, the results from domain engineer's perspective show that, in real-world evolution, it is more likely to extract functionality from a typically selected feature compared to random evolution operations.

Moreover, for some evolution operation instances, particular guidance elements cover no configuration, e.g., for $Delete_1$. However, for each guidance element, some evolution operations exist that result in many configurations being covered by that guidance element. For instance, the guidance element $Merge_0$ has covered only few configurations in general, but it covers **50**% of the configurations for one evolution operation for the *ERP* product line (cf. Fig. 7 bottom). Consequently, each guidance element is relevant.

To estimate the cost for application engineers to apply guidance, we analyze the automation degree (cf. RQ2a). Thus, for guidance elements with *automatic* type (cf. Sect. 3), no additional effort by application engineers is required as their configurations are updated automatically. For guidance with *semi-automatic* type, application engineers have to select only the update operation which fits best. For the *delete feature* template, we are able to automate guidance application ($Delete_0$) for between **18.8**% (cf. Fig. 7 bottom) and **100**% (cf. Figs. 6 and 7). For the *merge features* template, we are able to automate ($Merge_0$ and $Merge_1$) between **43**% (cf. Fig. 7 bottom) and **100**% (cf. Figs. 6 and 7 bottom) of the cases. For the *extract feature* template, we deliberately do not provide any automated guidance as new configuration options always arise, which may be of interest for application engineers.

For *semi-automatic* guidance, knowledge of both engineer roles is required. Thus, for all configurations covered by the guidance elements $Delete_1$, $Merge_2$, $Merge_3$, $Extract_0$, and $Extract_1$, application engineers must select an update operation. Regarding RQ2b, in the median for the *delete feature* and *merge feature* templates, knowledge of both engineers is required for few configurations. In the worst case, for **81.2**% ($Delete_1$, cf. Fig. 7 bottom) of the configurations, knowledge of both engineers is required. For the *extract* template, we need knowledge of both engineers for all configurations. The configurations for which knowledge of both engineers is required, are those configurations for which existing methods [16,20–22,29] do not suffice as they do not provide the possibility to share knowledge between the engineers. Even if the median values are not high, in some cases for up to **81.2**% of the configurations, application engineers are left alone in updating the configurations.

One goal of updating configurations to the new product-line version is to preserve behavior of the resulting products. Thus, in RQ2c, we are interested in the number of configurations for which product behavior can be preserved. The respective relevant guidance elements for the delete and merge templates are $Delete_0$, $Merge_0$, and $Merge_1$. This

Table 5 Comparison with partial refinement theory [39] (baseline)

Evolution operation	Perspective	Median configurations covered more than baseline (%)	Best case additionally covered configurations (%)
Delete	Domain engineer	2.1	27.0
	Application engineer	4.8	81.2
Merge	Domain engineer	1.8	26.0
	Application engineer	5.7	57.1
Extract	Domain engineer	0.0	0.0
	Application engineer	0.0	0.0

exactly matches the automation degree for those templates. Even if we do not provide full automated guidance for the extract new feature template, we are able to preserve product behavior for *all* configurations. Consequently, we are able to preserve product behavior for most of the configurations. Most of the configurations can remain as they were before evolution, i.e., $Delete_0$ and $Merge_0$. Thus, without our method, behavior would be preserved as well, but with our method, application engineers can be assured that they do not encounter unexpected behavior changes and know which evolution steps to focus on.

Existing methods to repair configurations do not consider changed product behavior but focus on reestablishing conformance of configurations with feature-model constraints [16,20,21,29], which can unexpectedly result in changed product behavior. With RQ2d, we address these cases, which occur for the merge features template for configurations covered by the guidance element $Merge_2$ and for the *extract new feature* template for configurations covered by $Extract_1$. Again, the median values are not high. However, in the worst case, product behavior changes for more than half of the configurations (**55.3**% for one extract evolution operation in the *ERP* product line), which would not be detected. Even if only few configurations fall into those categories, these are critical cases that can lead to severe problems and cost.

To set the results into perspective, we compare our methodology to the partial refinement theory [39] as a baseline. The partial refinement theory also considers product behavior after product-line evolution, but provides no support for cases in which no product preserves behavior. However, they do not consider the communication barrier between domain and application engineers. Thus, they do not provide concepts to enable application engineers to know how to update configurations to preserve product behavior nor to support cases in which no product behavior can be preserved.

Table 5 compares the results of our methodology with the partial refinement theory [39]. In particular, we highlight for which percentage of configurations our methodology additionally provides support. In the median, the percentage of additional configurations our methodology supports is rather low. However, if only $1,000$ configurations exist, this still means that in the median, we additionally support 18–57 configurations. In the best case, we additionally provide support for 81.2% ($Delete_1$, cf. Fig. 7 bottom) of the configurations. If these configurations had not been supported, this could lead to severe problems. A special case is the extract feature evolution operation. If this operation is performed, product behavior can always be preserved and, thus, the partial refinement theory [39] provides support for all configurations. However, our methodology goes beyond the partial refinement theory, and we support to *deliberately* update configurations such that product behavior is not preserved. For instance, if only the extracted functionality is required and the remaining functionality can be dropped. These cases are not represented by the numbers in Table 5 as it depends on the decision of application engineers which configuration update operation to choose. In summary, guided configuration evolution is relevant in many cases, especially if knowledge of both engineer roles is required. Additionally, our method provides awareness for application engineers regarding the product behavior preservation and our method is highly automated.

7.3 Threats to validity

Internal validity of the qualitative evaluation might be biased as we interviewed only one person who may have misinterpreted the evolution. However, this interview partner is a leading employee with deep knowledge of the product-line implementation which reduces the chance for mistakes. Additionally, in contrast to real-world application of our methodology, we as external researchers adapted and defined guidance. As discussed in Sect. 7.1, we expect domain engineers to be able to define and adapt guidance as they are already experts in defining feature models and variable implementations. Regarding the effort, we expect that engineers might take longer to specify the guidance (e.g., in a tool). However, in contrast to real-world engineers, we had to understand the project domain and features before defining guidance. Thus, we expect similar results. Nonetheless, we will evaluate the application of our method with a stable prototype of a tool suite by real-world domain engineers our future work.

The random generation in our quantitative evaluation may also affect internal validity. As we are not aware of any open-source product line for which commit history and configurations are publicly available, we decided to generate configurations for Linux and generate evolution scenarios for the *Agrib* and *ERP* product lines. Thus, parts of the evaluation use real-world product-line evolution and other parts use real-world configurations. Considering all configurations and all possible applications of templates is not feasible due to combinatorial explosion. However, this issue relates to the problem we address in this work: domain engineers do not know which configurations exist.

To reduce random bias, we heavily used repetitions by considering 1,000 configurations for each commit of the Linux kernel and 100 applications per each template for both *Agrib* and *ERP* product lines. The random generation of configurations affects only the evaluation of the Linux kernel but not the *Agrib* and *ERP* product lines. Random application of templates affects only the evaluation of the *Agrib* and *ERP* product lines but not the evaluation of the Linux kernel. As the results for both perspectives are similar, this increased confidence in our evaluation. To generate configurations, we used the random generator of FeatureIDE [35] (version 3.3) which does not generate uniformly distributed configurations. However, tools and methods to uniformly generate distributed configurations do not (yet) scale for large variability models as used in the evaluation [40]. Additionally, real-world configurations are not uniformly distributed and it is not possible to make statements about distribution without domain knowledge.

The tools we used for the quantitative evaluation may affect internal validity as they may contain defects. With *KConfigReader* and *FEVER*, we rely on tools that have been used in prior studies [13,22,31,33,41]. In particular, *FEVER* may detect all or may wrongly detect commits matching the templates in the history of the Linux kernel. *FEVER* missing commits matching operations in the history is uncritical as we were interested neither in all commits matching the operations nor in the probability of occurrence.

We increased confidence in the external validity with a combination of strategies. First, we analyzed a total of four real-world product lines from different domains. Second, we analyzed one open-source and three closed-source product lines with different implementation techniques. Thus, we reduced the threat that our method is applicable only to systems with a certain nature. Finally, we focused on evolution operations that have been identified as relevant in the literature [22,27–29] and which we indeed could confirm for Linux.

8 Related work

Highly configurable software systems and the evolution of such systems have been subject to recent research. Xu et al. [18] identified misconfigurations leading 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. [19] address a very similar problem as guided configuration evolution. They are interested in preserving product behavior after evolution by analyzing products' control flow. However, in contrast to Zhang et al. [19], we use a conservative and almost black-box approach. This method could be used complementarily by domain engineers if product behavior cannot be preserved to devise a suggestion for an update operation.

Recent research analyzed and categorized evolution of product lines and, in particular, the mapping between variability model and artifacts [27,31,42,43]. 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 [31]. We used FEVER to find and extract the commits of the Linux kernel for our evaluation (cf. Sect. 7.2.1). Passos et al. analyzed the evolution of the Linux kernel variability model and associated artifacts to extract evolution patterns [27]. We used some of these templates as motivating examples from real-world feature models. In the mentioned research [27,31,42] categorizations of commits are considered, but the guided configuration evolution is more generic and helps to update configurations. Ziegler et al. analyze the Linux kernel evolution and identify artifacts affected by changes to the variability model [43]. These results are incorporated in regression testing of configurations mapped to changed artifacts. Thus, they explicitly test artifacts affected by changes. However, they do not specify on how to fix configurations. Their approach could be improved incorporating product behavior preservation properties of evolution operations. Moreover, several authors identify dead or superfluous `#ifdef` blocks (i.e., feature-artifact mapping entries) [43–46]. Each analysis could be integrated with guided configuration evolution to check for new dead or superfluous mapping items after each evolution.

Other research defines refactorings for product-line evolution. Thüm et al. [47] and Alves et al. [48] classify feature-model evolution in terms of changes to the set of valid configurations. Both approaches focus on refactorings for feature models and do not consider product behavior of configurations. Schulze et al. define refactoring operations for product lines using feature-oriented and delta-oriented

programming [49,50]. Seidl et al. define evolution operations to co-evolve three spaces: feature models, artifacts, and mappings [51]. For operations affecting more than one space, they define how to co-evolve the other spaces [51]. 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 [24] and Neves et al. proposed several evolution templates preserving product behavior using this theory [28]. Sampaio et al. extended this theory by introducing partially safe evolution templates, preserving product behavior for a subset of configurations [22,39]. 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. [24], Neves et al. [28], and Sampaio et al. [22,39] 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 [16]. Semi-automatic approaches proposed either to provide the complete set of fixes with the smallest number of feature changes [21] or to gradually reach the desired fix using application engineers' feedback [20]. 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 feature-artifact mapping into account. Thus, the fixes may lead to different product behavior and, therefore, provide a false sense of correctness.

9 Conclusion

We presented guided configuration evolution, a methodology for automating the process of updating configurations semantically after product-line evolution as far as possible. 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 for a new product-line version while knowing the impact on product behavior. Even if it is impossible to communicate directly, our methodology allows 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 and show the relevance in our practical evaluation. To assess effectiveness, efficiency, and acceptance for real-world product-line evolution processes, we plan to perform a long-term study over several years with our industry partners. A second future work opportunity is an extension to our method that ensures configuration validity after applying configuration 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 change templates sustainably. 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.

Acknowledgements 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.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Thüm, T., Krieter, S., Schaefer,I.: Product configuration in the wild: strategies for conflicting decisions in web configurators. Proc. Configuration Workshop (ConfWS). pp. 1–8, RWTH Aachen University (2018)
2. Pett, T., Thüm, T., Runge, T., Krieter, S., Lochau, M., Schaefer, I.: Product sampling for product lines: the scalability challenge. Proc. Int'l Systems and Software Product Line Conf. (SPLC). pp. 78–83, ACM (2019)

3. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations. Springer, Principles and Techniques (2005)
4. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-oriented Software Product Lines: Concepts and Implementation. Springer, Berlin (2013)
5. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S. : Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute (1990)
6. Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K.: Software diversity: state of the art and perspectives. Int. J. Softw. Tools Technol. Transf. **14**, 477–495 (2012)
7. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. ACM/Addison-Wesley, New York (2000)
8. Quinton, C., Vierhauser, M., Rabiser, R., Baresi, L., Grünbacher, P., Schuhmayer, C.: Evolution in dynamic software product lines. J. Softw. Evolut. Process. **33**, 2 (2021)
9. Pett, T., Krieter, S., Runge, T., Thüm, T., Lochau, M., Schaefer, I.: Stability of product-line sampling in continuous integration. In Proc. int'l working conf. on variability modelling of software-intensive systems (VaMoS). ACM (2021)
10. Michelon, G.K., Obermann, D.,Linsbauer, L., Assunção, W.K.G., Grünbacher, P., Egyed, A.: Locating feature revisions in software systems evolving in space and time. In Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM (2020)
11. Ananieva, S., Greiner, S., Kühn, T., Krüger, J., Linsbauer, L., Grüner, S., Kehrer, T., Klare, H., Koziolek, A., Lönn, H., Krieter, S., Seidl, C., Ramesh, S., Reussner, R., Westfechtel, B.: A conceptual model for unifying variability in space and time. In Proc. Int'l systems and software product line conf. (SPLC). pp. 1–12, ACM (2020)
12. Kröher, C., Gerling, L., Schmid, K.: Identifying the intensity of variability changes in software product line evolution. In Proc. Int'l systems and software product line conf. (SPLC). pp. 54–64, ACM (2018)
13. Gomes, K., Teixeira, L., Alves, T., Ribeiro, M., Gheyi, R.: characterizing safe and partially safe evolution scenarios in product lines: an empirical study. In Proc. int'l workshop on variability modelling of software-intensive systems (VaMoS). ACM (2019)
14. Berger, T., Nair, D., Rublack, R., Atlee, J.M., Czarnecki, K., Wąsowski, A.: Three cases of feature-based variability modeling in industry. In Proc. int'l conf. on model driven engineering languages and systems (MODELS). pp. 302–319, Springer (2014)
15. Mukelabai, M., Nešić, D., Maro, S., Berger, T., Steghöfer, J.-P.: Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In Proc. int'l conf. on automated software engineering (ASE). pp. 155–166, ACM (2018)
16. White, J., Schmidt, D.C., Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated diagnosis of product-line configuration errors in feature models. In Proc. int'l systems and software product line conf. (SPLC). pp. 225–234, IEEE (2008)
17. Bosch, J.: Software product lines: organizational alternatives. In Proc. int'l conf. on software engineering (ICSE). pp. 91–100, IEEE (2001)
18. Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., Pasupathy, S.: Do no blame users for misconfigurations. In Proc. ACM symposium on operating systems principles (SOSP). pp. 244–259, ACM (2013)
19. Zhang, S., Ernst, M.D.: Which configuration option should I change? In Proc. int'l conf. on software engineering (ICSE). pp. 152–163, ACM (2014)
20. Wang, B., Passos, L., Xiong, Y., Czarnecki, K., Zhao, H., Zhang, W.: SmartFixer: fixing software configurations based on dynamic priorities. In Proc. Int'l systems and software product line conf. (SPLC). pp. 82–90, ACM (2013)
21. Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In Proc. int'l conf. on software engineering (ICSE). pp. 58–68, IEEE (2012)
22. Sampaio, G., Borba, P., Teixeira, L.: Partially safe evolution of software product lines. In Proc. int'l systems and software product line conf. (SPLC). pp. 124–133, ACM (2016)
23. Nieke, M., Sampaio, G., Thüm, T., Seidl, C., Teixeira, L., Schaefer, I.: GuyDance: guiding configuration updates for product-line evolution. In Proc. int'l workshop on variability and evolution of software-intensive systems (VariVolution). pp. 56–64, ACM (2020)
24. Borba, P., Teixeira, L., Gheyi, R.: A theory of software product line refinement. Theor. Comput. Sci. **455**, 2–30 (2012)
25. Thüm, T., Kästner, C., Erdweg, S., Siegmund, N.: Abstract features in feature modeling. In Proc. int'l systems and software product line conf. (SPLC). pp. 191–200, IEEE (2011)
26. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Am. Math. Soc. **74**(2), 358–366 (1953)
27. Passos, L., Teixeira, L., Dintzner, N., Apel, S., Wąsowski, A., Czarnecki, K., Borba, P., Guo, J.: Coevolution of variability models and related software artifacts. Empir. Softw. Eng. **21**, 4 (2016)
28. Neves, L., Borba, P., Alves, V., Turnes, L., Teixeira, L., Sena, D., Kulesza, U.: Safe evolution templates for software product lines. J. Syst. Softw. **106**, 42–58 (2015)
29. Nieke, M., Seidl, C., Schuster, S.: Guaranteeing configuration validity in evolving software product lines. In Proc. int'l workshop on variability modelling of software-intensive systems (VaMoS). pp. 73–80, ACM, (2016)
30. Owre, S., Rajan, S.P., Rushby, J.M. Shankar, N., Srivas, M.K.: PVS: combining specification, proof checking, and model checking. In Proc. Int'l conf. on computer aided verification (CAV). pp. 411–414, Springer (1996)
31. Dintzner, N., van Deursen, A., Pinzger, M.: FEVER: an approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. Empir. Softw. Eng. **23**(2), 905–952 (2018)
32. Krieter, S., Thüm, T., Schulze, S., Schröter, R., Saake, G.: Propagating configuration decisions with modal implication graphs. In Proc. int l conf. on software engineering (ICSE). pp. 898–909, ACM (2018)
33. Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation. In Proc. conf. on object-oriented programming, systems, languages and applications (OOPSLA). pp. 805–824, ACM (2011)
34. Montaghami, V., Rayside, D.: staged evaluation of partial instances in a relational model finder. In Proc. int'l conf. on abstract state machines, alloy, B, TLA, VDM, and Z. pp. 318–323, Springer (2014)
35. Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G.: Mastering Software Variability with FeatureIDE. Springer, Berlin (2017)
36. Pereira, J.A., Matuszyk, P., Krieter, S., Spiliopoulou, M., Saake, G.: A feature-based personalized recommender system for product-line configuration. In Proc. int'l conf. on generative programming and component engineering (GPCE). ACM (2016)
37. Pereira, J.A., Schulze, S., Figueiredo, E., Saake, G.: N-Dimensional tensor factorization for self-configuration of software product lines at runtime. In Proc. int'l systems and software product line conf. (SPLC). p. 87–97, ACM (2018)
38. Pereira, J.A., Schulze, S., Krieter, S., Ribeiro, M., Saake, G.: A context-aware recommender system for extended software product line configurations. In Proc. int'l workshop on variability modelling of software-intensive systems (VaMoS). p. 97–104, ACM (2018)

39. Sampaio, G., Borba, P., Teixeira, L.: Partially safe evolution of software product lines. J. Syst. Softw. **155**, 17–42 (2019)
40. Oh, J., Gazzillo, P., Batory, D.: t-Wise coverage by uniform sampling. In Proc. int'l systems and software product line conf. (SPLC). pp. 84–87, ACM (2019)
41. El-Sharkawy, S., Krafczyk, A., Schmid, K.: Analysing the kconfig semantics and its analysis tools. In Proc. int'l conf. on generative programming: concepts & experiences (GPCE). pp. 45–54, ACM (2015)
42. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex feature model differences. Automat. Softw. Eng. **23**(4), 687–733 (2015)
43. Ziegler, A., Rothberg, V., Lohmann, D.: Analyzing the impact of feature changes in linux. In Proc. int'l workshop on variability modelling of software-intensive systems (vamos). pp. 25–32, ACM, (2016)
44. Tartler, R., Lohmann, D., Sincero, J., Schröder-Preikschat, W.: Feature Consistency in Compile-time-configurable system software: facing the linux 10,000 feature problem. Proc. Europ. Conf. on Computer Systems. pp47–60 (2011)
45. Tartler, R., Lohmann, D., Dietrich, C., Egger, C., Sincero, J.: Configuration coverage in the analysis of large-scale system software. ACM SIGOPS Oper. Syst. Rev. **45**(3), 10–14 (2012)
46. Nadi, S., Dietrich, C., Tartler, R., Holt, R.C., Lohmann, D.: Linux variability anomalies: what causes them and how do they get fixed?," In Proc. working conf. on mining software repositories (MSR). pp. 111–120, IEEE (2013)
47. Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In Proc. int'l conf. on software engineering (ICSE). pp. 254–264, IEEE, (2009)
48. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., de Lucena, C.J.P.: Refactoring product lines. In int'l conf. on generative programming and component engineering (GPCE). pp. 201–210, ACM, (2006)
49. Schulze, S., Thüm, T., Kuhlemann, M., Saake, G.: Variant-preserving refactoring in feature-oriented software product lines. In Proc. int'l workshop on variability modelling of software-intensive systems (VaMoS). pp. 73–81, ACM, (2012)
50. Schulze, S., Richers, O., Schaefer, I.: Refactoring delta-oriented software product lines. In Proc. int'l conf. on aspect-oriented software development (AOSD). pp. 73–84, ACM, (2013)
51. Seidl, C., Heidenreich, F., Aßmann, U.: Co-evolution of models and feature mapping in software product lines. In Proc. int'l systems and software product line conf. (SPLC). pp. 76–85, ACM, (2012)

Michael Nieke is a post doc at IT University of Copenhagen, Denmark, with Prof. Christoph Seidl. From 2015 to 2021, he was Ph.D. student at TU Braunschweig, Germany, with Prof. Ina Schaefer. His research interests are modeling and analyzing highly configurable software systems, software evolution, and model-driven software engineering.

Gabriela Sampaio is a Ph.D. student at Imperial College London and a member of the Program Specification and Verification group. Her research interests include program verification, web development, software product lines, and software reuse.

Thomas Thüm since January 2020, he is a professor for the Construction and Analysis of Secure Software Systems at the University of Ulm in Germany. From 2015 to 2019, he was a post doc at the TU Braunschweig in Ina Schaefer's institute. He received his Ph.D. in 2015 from the University of Magdeburg under the supervision of Gunter Saake. His Ph.D. thesis received the Dissertation Award 2015 of the University of Magdeburg and his Master's thesis the Software Engineering Award 2011 of the Ernst Denert Foundation. He coauthored more than 100 peer-reviewed publications and is known for his contributions to the well-known open-source project FeatureIDE.

Christoph Seidl is an Assistant Professor for Software Engineering at IT University of Copenhagen, Denmark. His research interests are highly configurable software systems/variability modeling (e.g., Software Product Lines), software evolution, model-driven software engineering, language engineering, and software visualization. As part of his research, he has worked with companies, among others, in the automotive industry, aerospace engineering, and virtual reality.

Leopoldo Teixeira is an assistant professor at the Informatics Center (CIn) of the Federal University of Pernambuco, where he leads the Software Testing and Analysis Research group, and is affiliated with the Software Productivity Group. His main research interests involve the following topics and their integration: software product lines and configurable systems, software evolution, refactoring, formal methods, software testing, and mobile development.

Ina Schaefer is Professor of Software Engineering and Automotive Informatics at TU Braunschweig. Her research focus is quality assurance and correctness-by-construction engineering, particularly for variant-rich and evolving software systems. She has received her Ph.D. from TU Kaiserslautern in 2008 and been a post doc at Chalmers University, Gothenburg.