Guaranteeing Configuration Validity in Evolving Software Product Lines

Michael Nieke, Christoph Seidl, Sven Schuster

Technische Universität Braunschweig
38106 Braunschweig, Germany
{m.nieke, c.seidl, s.schuster}@tu-braunschweig.de

ABSTRACT

Software Product Lines (SPLs) are an approach to capture families of closely related software systems in terms of commonalities and variabilities where individual variants are defined by configurations of selected features. Specific (partial) configurations may be of particular importance to SPL manufacturers, e.g., if they are very popular or used by major customers. SPLs are subject to evolution, which may inadvertently break existing configurations, e.g., if a previously selected feature does no longer exist. This is problematic as it may delay or completely prevent creation of previously existing important variants causing monetary loss and customer dissatisfaction. In this paper, we present a method to *lock* specific configurations to ensure their validity during evolution of the SPL. For this, we present Temporal Feature Models (TFMs) and dedicated evolution operations as a semantic-enriched first-class notion for evolution of feature models, which we use to assess the impact on existing configurations. Using the presented method, it is possible to guarantee that locked configurations remain valid during SPL evolution and make statements on which part of the evolution would break the configurations.

CCS Concepts

•Software and its engineering → Software notations and tools; Software evolution;

Keywords

Temporal Feature Model (TFM), Software Product Line (SPL), Evolution, Configuration

VaMoS '16, January 27 - 29, 2016, Salvador, Brazil

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4019-9/16/01…$15.00

DOI: http://dx.doi.org/10.1145/2866614.2866625

1. INTRODUCTION

Software Product Lines (SPLs) are an approach for systematic reuse for closely related software systems in terms of commonalities and variabilities [15]. The variability model, e.g., a Feature Model (FM) [12], is defined in the *problem space*, whereas the realization artifacts, such as code or models, are defined in the *solution space* [6]. *Configurations* bind the variability in the problem space, e.g., by selecting a set of features, to define the conceptual part of a concrete *variant* of the SPL. *Partial configurations* bind a subset of the possible variability leaving some variability decisions open. Configurations are *valid* if they are consistent with the configuration rules defined in the variability model. The *variability space* contains all possible variants, which can be created with the respective SPL.

SPL developers typically have several particularly important configurations, which are popular or used by major customers. However, these configurations are often partial as SPL manufacturers want to leave certain variability decisions to customers. These major (partial) configurations need to be handled prioritized, as breaking these configurations may lead to delays in production which potentially results in monetary loss. Also, a break in configuration may possibly lead to a loss of trust in the manufacturer.

As all software systems, SPLs need to evolve due to changed or new requirements [11]. This evolution may also affect the FM when restructuring the problem space. Thus, features may be added or removed and constraints may change the variability space. However, evolution can break configurations so that variants based on these configurations can no longer be generated. This is particularly critical for particularly important (partial) configurations of SPL developers. For example, a major customer of a computer system manufacturer may have bought many computers of one configuration type. All these computers run on a Linux operating system and have a specific USB controller. However, as the Linux kernel is evolving as well, the driver for that specific USB controller may have been dropped in the new version of the Linux kernel [14]. As a consequence, the configurations using this specific USB controller get invalidated due to this evolution. Thus, the Linux kernel on the computers of the major customer cannot be updated to new versions and potential security flaws of the Linux kernel cannot be fixed. Therefore, the customer has to buy new computers without that USB controller or has to remain with vulnerable systems. This could lead to monetary loss or security issues and, potentially, customer dissatisfaction with the manufacturer. Hence, for SPL manufacturers, it is crucial to prevent that selected particularly important (partial) configurations are invalidated during evolution.

In this paper, we present a methodology to guarantee the validity of specific configurations during evolution through *configuration locking*. To achieve this goal, we introduce Temporal Feature Models (TFMs) as a methodology to model FMs and their evolution as first-class entities. On this basis, we analyze the impact of evolution on existing (partial) configurations and define categories for *broken configurations*. Finally, we prohibit evolutions that would break locked configurations so that their validity can be guaranteed.

The rest of the paper is structured as follows: In Section 2, we introduce fundamental principles our methodology is built on. In Section 3, we present Temporal Feature Models (TFMs) as methodology to capture evolution of an FM as first-class entity along with *evolution operations*. In Section 4, we elaborate on analyses to consider the impact of evolution on existing configurations and introduce the concept of *locking* configurations. In Section 5, we demonstrate the applicability with a case study. In Section 6, we discuss related work. Finally, in Section 7, we close with a conclusion and an outlook to future work.

2. BACKGROUND

Feature Models (FMs) capture the variability of a system in terms of hierarchically structured features – units of reuse representing configurable functionality [12]. Features may be *optional* or *mandatory* but may only be selected if their parent is selected. Additionally, features are organized in *unbounded*, *or* and *alternative* groups. To express the types of features and groups, *cardinality-based feature models* assign a lower and upper cardinality to each feature and group [8]. A cardinality of (0..1) represents an *optional* feature and a cardinality of (1..1) a *mandatory* feature. For single features, we explicitly do not support cardinalities greater than 1, which would result in cloned features. The cardinality for *alternative* groups is (1..1) so that the contained features are mutually exclusive but at least one feature has to be selected. *Or* groups containing n features have a cardinality of $(1..n)$ implying the selection of at least one feature of the group. For *unbounded* groups containing n features, the cardinality is $(0..n)$, which does not restrict the selection of features.

Feature attributes may be used to provide additional variability for features [3] that goes beyond mere (de)selection. For example, the language for the user interface of an infotainment system. A feature may have an arbitrary number of attributes, which consist of an identifier and a type. Attributes may be incorporated in the configuration process by setting an appropriate value for a dedicated attribute of the respective feature. Within our work, we assume that each attribute has a default value.

Additionally, *cross-tree constraints* on features can be specified via propositional formulas. Such constraints can define dependencies between features that are not defined by the structure of the FM. To support constraints on feature attributes, these formulas can be extended by Boolean expressions on feature attributes, e.g., limiting the range of an Integer attribute.

Configurations are used to derive concrete variants of an SPL. In configurations, features of the FM may be selected. Moreover, with feature attributes, values for the attributes of each selected feature may be set. If no value is defined, the default value of the attribute is assumed. A configuration is considered *valid* when the feature selection is compatible with the structure of the feature model and all of its cross-tree constraints and when the set values obey the type of their respective attribute. The *variability space* contains all possible configurations.

Figure 1 depicts an excerpt from our case study used as running example where the configuration options of a `Car` with its `Assistance Systems` and `Infotainment System` are represented as FM. The original FM in Figure 1a is evolved by adding and deleting features as well as restructuring the existing features to the FM depicted in Figure 1b. In the following, we elaborate on the respective modifications to demonstrate our individual contributions.

3. EVOLUTION OF FEATURE MODELS

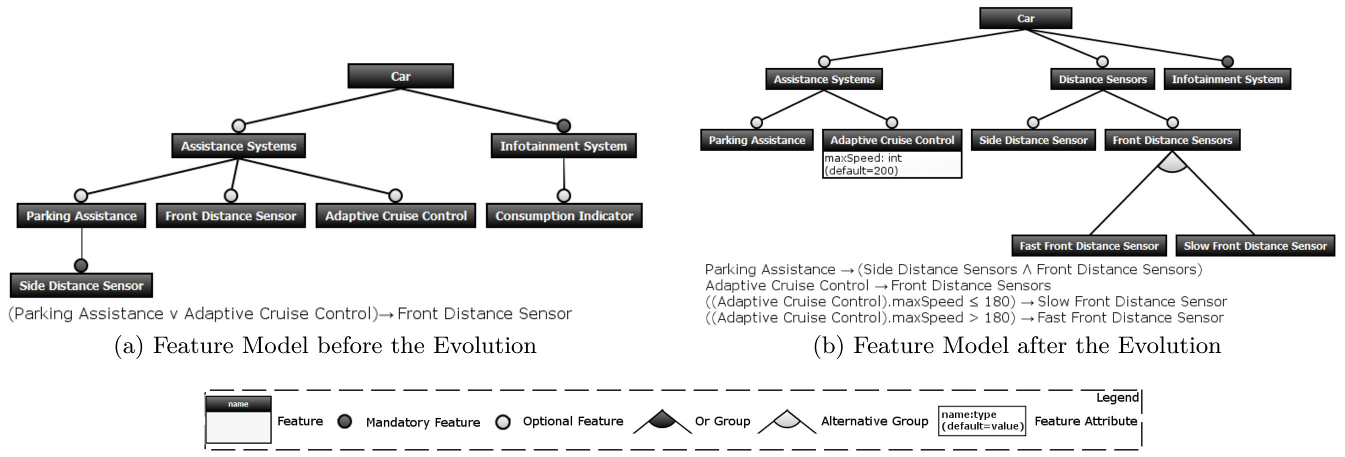
Evolving SPLs include the evolution of the FM. Modifying the FM without keeping track of the evolution itself results in loss of information as the old versions of the FM cannot be retrieved. Keeping old versions of an FM can be necessary, e.g., to support customers with products based on old versions of the FM with updates. Moreover, commonly there is no formal documentation on the evolution of the FM, i.e., there is no semantics, providing information about how and why the FM evolved, which can be of relevance for reasoning about the evolution. For example, if a feature is renamed and no documentation on the renaming is stored, it is similar to deleting the old feature and adding a new feature with the new name. To keep track of the evolution and to preserve the old versions of the FM in one model, we need a first-class notion of evolution. Based on the provided information of the first-class evolution, we present how to attribute additional semantics to evolutions by extracting evolution operations from the timespan an element is temporally valid.

3.1 Temporal Feature Models

To capture evolution of FMs and preserve the information of the old model, we present *Temporal Feature Models (TFMs)*. TFMs define the concept of *evolving elements*, capturing evolution as first-class entities. Evolving elements have a limited timespan in which they are temporally valid.

Their temporal validity ϑ is an interval defined by two points in time: the start of their temporal validity, ϑ_{since}, and the end, ϑ_{until}. The temporal validity is defined as $\vartheta = [\vartheta_{since}, \vartheta_{until})$, meaning that the element is not valid at ϑ_{until}. This is necessary to provide seamless temporal validities of elements, e.g., if two elements e_1, e_2 have $\vartheta_{e1until} = \vartheta_{e2since}$ means that for each point in time between $\vartheta_{e1since}$ and $\vartheta_{e2until}$ exactly one of these elements is valid. In Figure 2, an example of an FM with different temporal validities of features is depicted. As can be seen, `F1` and `F2` are valid since t_1. However, `F1` is only valid until t_2, whereas `F3` is valid since t_2. As a result, at point t_1, the features `F1` and `F2` are valid, whereas features `F2` and `F3` are valid at point t_2 and t_3. Note that deleting an element means setting ϑ_{until} to the the desired deletion date.

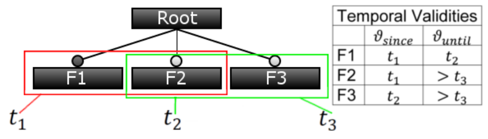
To allow arbitrary evolution of an FM, it has to be possible to represent changes on each individual element making evolution itself a first-class entity. As features are not only removed and added but groups and features can be moved, their type can be changed or they can be renamed, the respective affected elements need to be modeled as first-class entities, too. In standard FMs, not all the information possibly affected by evolution is captured in dedicated elements


Car
Assistance Systems
Infotainment System
Parking Assistance
Front Distance Sensor
Adaptive Cruise Control
Consumption Indicator
Side Distance Sensor
(Parking Assistance ∨ Adaptive Cruise Control)→ Front Distance Sensor
Car
Assistance Systems
Distance Sensors
Infotainment System
Parking Assistance
Adaptive Cruise Control
maxSpeed: int (default=200)
Side Distance Sensor
Front Distance Sensors
Fast Front Distance Sensor
Slow Front Distance Sensor
Parking Assistance → (Side Distance Sensors ∧ Front Distance Sensors)
Adaptive Cruise Control → Front Distance Sensors
((Adaptive Cruise Control).maxSpeed ≤ 180) → Slow Front Distance Sensor
((Adaptive Cruise Control).maxSpeed > 180) → Fast Front Distance Sensor
Legend
name
Feature
Mandatory Feature
Optional Feature
Or Group
Alternative Group
name:type (default=value)
Feature Attribute

(a) Feature Model before the Evolution

(b) Feature Model after the Evolution

Figure 1: Exemplary Feature Model of a Car



	ϑ_{since}	ϑ_{until}
F1	t_1	t_2
F2	t_1	$> t_3$
F3	t_2	$> t_3$

Figure 2: Example of Temporal Validities of Features

but some of it is represented by relations between elements. For example, groups are related to a parent feature, which determines their location in the FM. In addition, groups are composed of an arbitrary set of features. Thus, the location of features in the FM is determined by their group membership. To allow evolution of group-parent feature relations and group compositions, it is necessary to capture temporal validities along with the affected relations. Hence, the relations of standard feature models become association classes in TFMs, which makes them first-class entities. In the following, we briefly elaborate on all first-class entities of TFMs:

- *Features* and *groups* to support their creation and deletion.
- *Cardinalities of features and groups* to support the evolution of types of features and groups.
- *Names of features* to support the renaming of features.
- *Attributes of features* to support their creation, renaming and deletion.
- *Group compositions* to support the relocation of features and changeable compositions of groups.
- *Feature children* to support the relocation of groups.
- *Cross-tree constraints* to support their creation and deletion.

Note that for each group exactly one *group composition* must exist for every point in time the group is temporally valid to prevent ambiguity. The same applies for *feature children*. For each group, exactly one *feature child* must exist, for every point in time the group is temporally valid. Moreover, TFMs explicitly do not support the evolution of sub-expressions of cross-tree constraints. However, evolving constraints is possible by invalidating the old constraint and adding a new one.

3.2 Evolution Operations

To determine the impact of FM evolution on configurations, it is necessary to reason about the semantics of that respective evolution. While temporal validities may be used uniformly to represent the evolution of feature model elements and do not require to record the specific type of modification during evolution, it is beneficial for analyses to have information on the nature of an applied change. For this purpose, we introduce *evolution operations* that can be derived from changes to temporal validities in order to attribute additional semantics to evolution. Through this, we can make more precise statements on why and through which operation the FM evolved as opposed to merely observing the effects of evolution as with diffs on FMs. We define five *atomic evolution operations*: *add*, *delete*, *rename*, *change type* and *move*. These operations are possibly applicable for different elements, e.g., the *move* operation can be applied for features and groups. More complex evolution operations may be defined by combining these operations.

The atomic evolution operations can be extracted by merely analyzing changes to the temporal validities of respective evolving elements. For this, we analyze the temporal validities, ϑ_{since} and ϑ_{until}, of the elements for two points in time, τ_1 and τ_2 ($\tau_1 \leq \tau_2$), for which the operations should describe the evolution. In the following, we describe the atomic evolution operations with the running example of Figure 1. In Table 1, the properties of the temporal validities for the respective evolution operations are depicted.

From the evolution of the running example, several *add* evolution operations can be extracted. The features `Front Distance Sensors` and `Fast Front Distance Sensor` are added. The *delete* operation is the opposite of the add operation and can be seen for feature `Consumption Indicator`. As the `Side Distance Sensor` is moved under `Distance Sensors`, its type has to be changed to optional. A *change type* operation can be extracted by analyzing the temporal validities of the cardinalities $card_1$ and $card_2$ of a feature or a group. As the `Adaptive Cruise Control` needs differ-

Evolution Operation	**Properties of Temporal Validities between two points in time τ_1, τ_2**
add	$\tau_1 < \vartheta_{since} \leq \tau_2 < \vartheta_{until}$
delete	$\vartheta_{since} \leq \tau_1 < \vartheta_{until} \leq \tau_2$
change type	$card_1 \neq card_2$, $\vartheta_{card_1 since} \leq \tau_1 < \vartheta_{card_1 until} \leq \vartheta_{card_2 since} \leq \tau_2 < \vartheta_{card_2 until}$
rename feature	$name_1 \neq name_2$, $\vartheta_{name_1 since} \leq \tau_1 < \vartheta_{name_1 until} \leq \vartheta_{name_2 since} \leq \tau_2 < \vartheta_{name_2 until}$
move feature	$group_1 \neq group_2$, $\vartheta_{compo_1 since} \leq \tau_1 < \vartheta_{compo_1 until} \leq \vartheta_{compo_2 since} \leq \tau_2 < \vartheta_{compo_2 until}$
move group	$parentFeature_1 \neq parentFeature_2$, $\vartheta_{featureChild_1 since} \leq \tau_1 < \vartheta_{featureChild_1 until} \leq \vartheta_{featureChild_2 since} \leq \tau_2 < \vartheta_{featureChild_2 until}$

Table 1: Evolution Operations and Temporal Validities

ent `Front Distance Sensors` for different maximum speeds, the `Front Distance Sensor` in Figure 1a is renamed to `Slow Front Distance Sensor` in Figure 1b to better distinguish it from the newly added sensor. Thus, a *rename feature* operation can be extracted. Within a diff, renaming a feature would be similar to deleting the old feature and adding a new one with the new name but with TFMs, we are able to detect that it is still the same feature. A *move feature* operation can be extracted if the feature is part of two group compositions, $compo_1$ of $group_1$ and $compo_2$ of $group_2$. The *move group* operation can be extracted equivalently to the *move feature* operations, whereas a group is related to multiple *feature child* entities with different parent features. In the example, the FM is restructured and a new feature `Distance Sensors` is introduced with all distance sensors as children. Thus, the `Side Distance Sensor` has to be moved under the new `Distance Sensors` feature.

With the atomic evolution operations, we are able to attribute semantics to the evolution of the FM. Thus, we can use these semantics to reason about the evolution of the FM.

4. ENSURING VALID CONFIGURATIONS

Configurations bind all variability of an SPL on a conceptual level by selecting a valid subset of all features that satisfies all configuration rules of an FM. In contrast, *partial* configurations leave some configuration choices open by binding only a part of the variability of the SPL. This can be done by selecting features to express that a certain feature always must be integrated in configurations which are based on the respective partial configurations. However, with our methodology, developers should also be able to explicitly exclude variability decisions with partial configurations. To this end, we extend the notion of a configuration to not just allow selecting features but to also explicitly support deselecting features.

During FM evolution, it may be inevitable that existing (partial) configurations get invalidated, e.g., by removing a feature that was selected within the configuration. However, some configurations may be of particular importance so that they specifically need to be maintained throughout evolution. For example, breaking configurations demanded by major customers may lead to monetary loss for the customer and, consequentially, reduced trust in the SPL maintainer. To prevent breaking these configurations, one solution is to prohibit evolution which leads to the invalidation. To this end, analyses making statements on the impact of evolution on configurations facilitate prohibiting such evolution. Thus, in the following sections, we introduce such analyses based on the semantic information provided by evolution operations. Additionally, our analyses are applicable for partial configurations, too. We then introduce *configuration locking* as concept to guarantee the validity of configurations by using the results of the impact analyses.

4.1 Change Impact Analyses

Inconsistencies caused by the evolution of FMs can arise easily. For example, a configuration based on the FM before the evolution in Figure 1a with `Assistance Systems`, `Parking Assistance`, `Front Distance Sensor` and `Side Distance Sensor` selected, would be inconsistent after the evolution of the example. After the evolution, the three features are still selected, although `Parking Assistance` is renamed to `Parking Pilot`. As the renaming is a refactoring and the feature remains the same, it is still selected. However, without TFMs but with FM diffs, the feature would not be selected anymore, as it would seem that the original feature had been deleted and that the renamed feature was a new one. However, considering the FM after the evolution, the feature `Side Distance Sensor` is still selected, but its new parent `Distance Sensors` is not selected. Thus, the configuration is not consistent as `Parking Pilot` cannot be selected without `Side Distance Sensor` and the sensors cannot be selected without `Distance Sensors`. However, evolution of SPLs is necessary and it may be hard to tell which part of the evolution broke a configuration, especially if many configurations can be affected.

To detect inconsistencies which may arise due to FM evolution, the impact of an evolution on configurations needs to be analyzed. Additionally, information on how the evolution affected certain configurations may be valuable for SPL developers. However, reasoning about the entire FM and every configuration may be needlessly time consuming when only specific configurations or individual evolution operations are of interest. Thus, we use the information captured within TFMs for analyses on the impact of evolutions on configurations. To prevent superfluous analyses, we only analyze selected and deselected elements of the respective configuration which changed during FM evolution. For this, we assume that all analyzed configurations were valid with respect to the FM before the respective evolution. Additionally, to give more detailed information on the impact of the evolution on a configuration, we provide *broken configuration categories*, which define in what way a configuration is affected by the evolution. We attribute each evolution to one of the following categories regarding its potential impact on a configuration:

- **Not Broken**: The configuration is not broken, but the following distinctions are made to provide warnings to SPL developers about possible side effects:
  - **Refactoring**: The evolution is a refactoring, and thus, has no impact on the variability space. For example, a feature is renamed.
  - **Unaffected**: The configuration is unaffected as the variability space is extended or the configu-

Add and Delete Operations	**Add**	**Delete**
Feature / Attribute	Extended	Outdated (if defined)
Group without contained Features	Unaffected	Unaffected
Constraint	Conflicting	Unaffected

Change Type Operation From → To	**Broken Category**
Mandatory → Optional	Unaffected
Optional → Mandatory	Conflicting (if deselected)
Alternative → Or	Unaffected
Or → Alternative	Conflicting (if more than one feature selected)
Unbounded → Or	Conflicting (if no feature selected)
Unbounded → Alternative	Conflicting (if not exactly 1 feature selected)
Or / Alternative → Unbounded	Unaffected

Move Operation	
Move Group to other parent	Conflicting (if new parent is not selected)
From Alternative	Conflicting (if feature selected)
To Alternative	Conflicting (if feature selected)
From Or	Conflicting (if only selected feature in group)
From Unbounded	Unaffected
To Unbounded / Or	

Rename Feature / Attribute	Refactoring

Table 2: Broken Categories for Evolution Operations

ration defines none of the removed variants. For example, the type of a selected feature is changed from optional to mandatory.

  - **Extended**: The variability space is extended and new elements are available. For example, a new feature is added. The SPL developers are informed that new variability is available.

- **Outdated**: The configuration is outdated as it contains elements that are not valid anymore. For example, a selected feature was deleted.

- **Conflicting**: The contained elements of the configuration are not consistent with the FM anymore. For example, a new constraint prohibits the simultaneous selection of two selected features.

To make more precise statements about evolution breaking configurations, we assign each evolution operation to one of the broken categories. With that, we can determine exactly which part of the evolution has broken a configuration. In Table 2, we assign the broken categories to the different atomic evolution operations. Note that we split up some operations as these distinctions allow us to provide a more precise categorization. If the categorization is annotated with a condition in brackets, the respective operation could also be categorized as *unaffected* if the condition is not true. Constraints are only evaluated if they affect any of the selected or deselected elements of the configurations. Adding and deleting a constraint may also result in an *unaffected* category if the constraint is already covered by another constraint. With this categorization, we can analyze the impact of FM evolution on configurations. Our methodology is applicable for partial configurations, too, as we analyze the consistency of the evolution only with respect to the elements defined in the configurations.

A combination of evolution operations does not necessarily result in the worst case broken category. For example, if a selected feature is moved into an alternative-group and the type of this group is changed to an or-group, the configuration is still valid although the categorization of this move operation would be *conflicting* for every configuration. Thus, more complex and composite evolution operations can be categorized differently from their constituent atomic evolution operations.

4.2 Configuration Locking

SPL developers may have several configurations of high importance, which should not get broken. However, evolution can lead to breaking existing configurations. To guarantee the validity of configurations, we introduce the concept of *configuration locking*. Thus, a locked configuration is a configurations that may never be broken. This means that all selected and deselected features, as well as defined attribute values, must remain consistent with the evolved FM. For this, the evolution of the FM is analyzed in respect to all *locked* configurations with the previously introduced change impact analyses. Thus, the evolution must not fall into the categories *outdated* or *conflicting* for each *locked* configuration. If developers try to evolve an FM and a locked configuration gets broken, the evolution cannot be performed. Additionally, we can tell why an evolution cannot be performed. Considering an approach, where the configuration and the new FM are given to an evaluator which checks the compatibility between the configuration and the FM, it is not possible to make statements about which part of the evolution has broken the configuration. Additionally, it is inevitably necessary to verify the entire FM and configuration. With our methodology, we can only check the parts of the FM and configuration that are part of the evolution. As composite evolution operations may only have a less strict categorization (e.g., unaffected instead of conflicting), the categories for composite evolution operations may only be stricter than they would have to be. Thus, the validity of the locked configuration is always guaranteed. Additionally, the SPL developers get informed if an evolution was categorized as *extended* or *unaffected* for any *locked* configuration so that they can review the respective evolution.

5. CASE STUDY

To demonstrate the benefits of configuration locking and to show the feasibility of our methodology, we provide a case study based on a fictitious but realistic scenario. To this end, we provide an exemplary implementation of TFMs, which is used in an artificial case study. We show that locking guarantees the validity of specific configurations during evolution.

5.1 Implementation

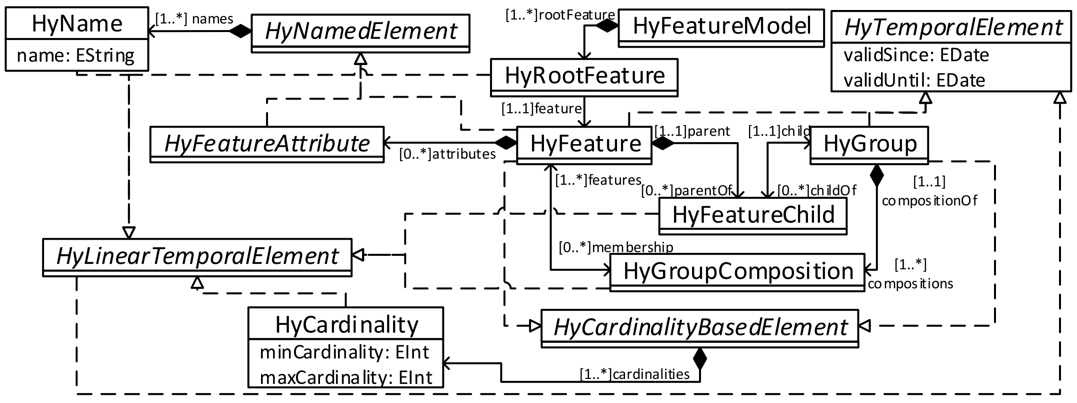


Figure 3: Metamodel for Temporal Feature Models (TFMs)

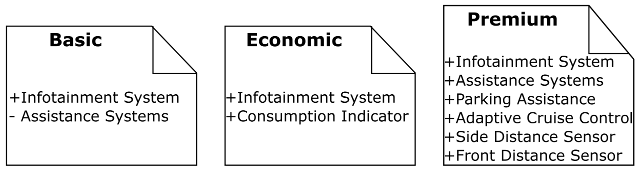
We implemented TFMs with the Eclipse Modeling Framework (EMF)[1]. For this, we created an Ecore meta model, which can be seen in Figure 3. Two interfaces are responsible for allowing the evolution of the elements. `HyTemporalElement` defines two fields: `validSince` and `validUntil`, which represent the previously introduced temporal validity ϑ. The `HyLinearTemporalElement` inherits from `HyTemporalElement` but specifies an additional superseding relation. This is used to express that a temporally valid element for a certain relation must exist for every relevant point in time. For example, the `HyCardinality`, representing the cardinality of a group or a feature, is a `HyLinearTemporalElement`. Thus, for a group or a feature, there must be exactly one valid cardinality for every point in time within the temporal validity of the feature or group. However, as can be seen, a `HyFeatureModel` consists mainly of `HyFeatures` and `HyGroups`. The relation for the root feature, as well as the composition of the groups and the parent feature of a group, are modeled as first-class entities to support the evolution of these relations. Thus, e.g., a group can have multiple `HyGroupCompositions`, as different compositions may be valid at different points in time. Moreover, each `HyFeatureAttribute` and each `HyName` has a validity, too. The complexity of the meta model is hidden from end users as models are represented in a graphical language similar to that used in Figure 1. For the end user, the only new concept is to provide temporal validities for the respective elements. To allow inspecting the temporal validity of the elements in a TFM at a selected point in time, we support a mode in which only the currently valid FM for that point in time is viewed.

A `HyConfiguration` inherits from `HyTemporalElement` and, therefore, has a temporal validity, too. It consists of `HyConfigurationElements`, which define the selection or deselection of a feature or a value for an attribute. However, `HyFeatureSelection` and `HyAttributeValueAssignment` inherit from `HyTemporalElement`, as the selection of features and the value assignment for attributes can evolve, too.

5.2 Case Study Scenario

Using this implementation, we perform a case study on a fictitious but realistic scenario to show the feasibility of modeling FM evolution with TFMs and analyzing the impact of such evolution on configurations. For this, we use the FMs of Figure 1, whose evolution is comprised of multiple operations. The original FM and its evolution are implemented as instance of the respective meta model.

As our focus is on the locking of configurations and, thus, on the impact of FM evolution on configurations, we provide three different configurations for the FM of Figure 1a, which can be seen in Figure 4. The explicitly selected features are depicted with a "+", whereas deselected features are depicted with a "-".



(a) (b) (c)

Figure 4: Exemplary Configurations for a Car

The *Basic* configuration only selects the mandatory `Infotainment System` and explicitly deselects the `Assistance Systems`. For economic drivers, the *Economic* configuration also selects the `Consumption Indicator`. As economic driving is very popular and customers demand an economic variant of the car, the manufacturer decides to *lock* the *Economic* configuration. The full featured *Premium* configuration selects every possible features of the original FM.

We analyze the impact of the evolution on these configurations with regard to the broken categories. As a new feature and a new attribute are added, the evolution falls into the *extended* category for all configurations. Thus, a person interested in maintaining a configuration could be notified that additional elements are available. However, as only specified elements need to be analyzed for the other categories, the *Basic* configuration, which only selects `Infotainment System`, is not affected by any evolution operation. Thus, *Basic* falls into the *unaffected* category. The *Economic* configuration selects additionally the `Consumption Indicator`. As this feature is deleted (respectively invalidated), it falls into the *outdated* category. For the *Premium* configuration, several operations were applied. The selected `Front Distance Sensor` is renamed to `Fast Front Distance Sensor`. As this is a *refactoring* and the reference to the object still exists, the con-

[1]https://eclipse.org/modeling/emf/

figuration is not yet broken. Additionally, `Front Distance Sensor` and `Side Distance Sensor` are moved underneath the new feature `Distance Sensors`. As the new parent is not selected, these evolution operations fall into the *conflicting* category. Additionally, the `Adaptive Cruise Control` receives a new attribute `maxSpeed`. This would fall into the *extended* category but, combined with the new constraints, it falls into the *conflicting* category. As the value for `maxSpeed` is not set, the default value of 200 is assumed. However, the constraint requires selecting `Fast Front Distance Sensor` if `maxSpeed` is greater than 180. As the *Economic* configuration has been *locked* and the evolution falls into the *outdated* category in respect to the *Economic* configuration, the SPL developer receives a notification about the evolution breaking the *locked* configuration. As result, the evolution could not be applied. If the *Premium* configuration would have been *locked* instead of the *Economic* configuration, this evolution scenario could still not have been applied regardless, as it would fall in the *conflicting* category in respect to the *Premium* configuration. Whereas, when only *locking* the *Basic* configuration, the evolution could be applied.

We realized the presented scenario of the case study with our methodology. With TFMs, we were able to capture variability in cardinality-based FMs with feature attributes while modeling their evolution as first-class entities at the same time. The presented analyses and *broken categories* allowed us to make statements concerning the impact of the evolution on the configurations. We showed that the concept of *locking* configurations provides guarantees to SPL developers as the respective configurations remain valid. However, with additional tool support, it would be easier to create the TFM models as it is complex to create instances due to the increased number of first-class entities, e.g., names and group compositions.

6. RELATED WORK

Different authors worked on various combinations of FMs and evolution. Seidl et al. [18, 19] capture versions of solution space artifacts with Hyper Feature Models (HFMs). In contrast, our methodology addresses evolution of the FM itself. Quinton et al. [16] mainly consider the impact of evolution on cardinality consistencies and, thus, the consistency of the FM. Gamez et al. [10] automatically create new configurations of evolved FMs and they measure the change impact by means of a difference between the old configuration and the new configuration. However, they do not explicitly define the evolution of an FM and their approach does not consider evolution as first-class entity. White et al. [21] formalize FM edits happening during the configuration finding process. They represent the FM as a Constraint-Satisfaction-Problem (CSP) and model arbitrary FM evolutions via added or removed constraints. However, they do not analyze which existing configurations are broken, do not support partial configurations and do not provide a first-class notion of evolution.

Botterweck et al. [5, 4] introduce *EvoFM* and consider evolution as first-class entity. However, they require the different evolution operations to be modeled explicitly whereas we can extract evolution operations from temporal validities of different elements. Our methodology offers the same level of semantics as *EvoFMs* but is more flexible in application as evolution operations do not need to be recorded. Additionally, to analyze an FM for a particular point in time, *EvoFMs* need to apply the sequence of all evolution operations to the respective point in time. Our methodology allows to select the valid elements of that point in time directly, which is particularly beneficial for long evolution histories.

Furthermore, there is work on analyzing the impact of evolution/modification of an FM on the variability space. Thüm et al. [20] provide four categories, which define how modifications of an FM influence the variability space. Neves et al. [13] describe a set of evolution templates called "safe evolutions" as they do not reduce the variability space. However, even evolutions that reduce the variability space may be safe regarding certain configurations. Reuling et al. [17] compare two versions of an FM and assign *Mutation Operators* to the extracted evolution. However, they do not analyze the impact of evolution on certain configurations.

Dintzner et al. [9] present an approach to analyze the impact of FM evolution on Multi-SPL configurations by determining the impact on "shared features", which are used by multiple sub-SPLs. In contrast, our methodology analyses the impact of evolution independently of SPLs boundaries, e.g., if a configuration contains multiple features from different sub-SPLs.

Batory [2] uses Logic Truth Maintenance Systems, to continually check the consistency of certain feature selections to a given FM. However, this methodology does not analyze explicitly the impact of evolution on configurations. Moreover, as only Boolean predicates are allowed, it does not suffice for attributed FMs.

Acher et. al [1] present an approach to provide additional semantics to FM modifications. However, they create a diff between two versions of an FM. This limits the semantics which can be provided for certain changes.

Furthermore, all the discussed impact analyses are not able to handle partial configurations. With the analyses provided in this paper, it is possible to handle complete and partial configurations alike.

7. CONCLUSION

In this paper, we presented a methodology to guarantee the validity of specific configurations during evolution of an FM. We introduced Temporal Feature Models (TFMs) to enable modeling of arbitrary evolution of attributed cardinality-based FMs by creating first-class entities for all elements subjected to evolution and assigning temporal validities to them. By extracting evolution operations from the temporal validities, we attribute additional semantics to the modeled evolution. On this basis, we measure the impact of FM evolution on (partial) configurations by only inspecting the FM parts relevant for the configuration. Furthermore, we introduce the concept of *configuration locking* to provide guarantees that the evolution of the FM does not break specific configurations.

In our future work, we will investigate the benefits of our incremental analyses compared to non-incremental analyses. As our analyses already support partial configurations, we intend to provide support for staged configuration [7] and, thus, to analyze which stages are affected by the evolution. Moreover, as we attribute additional semantics to individual evolutions by extracting evolution operations, we are planning to create a methodology to provide *semantic* repair mechanisms of configurations. Finally, we intend to perform an industrial case study to analyze the suitability of our methodology in a realistic environment.

Acknowledgments

The authors would like to thank our colleague Thomas Thüm and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the European Commission within the project HyVar (grant agreement H2020-644298).

8. REFERENCES

[1] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. Feature model differences. In J. Ralyt'e, X. Franch, S. Brinkkemper, and S. Wrycza, editors, *Advanced Information Systems Engineering*, volume 7328 of *Lecture Notes in Computer Science*, pages 629–645. Springer Berlin Heidelberg, 2012.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 491–503, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski. Evofm: Feature-driven planning of product-line evolution. In *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*, PLEASE '10, pages 24–31, New York, NY, USA, 2010. ACM.

[5] G. Botterweck, A. Pleuss, A. Polzer, and S. Kowalewski. Towards feature-driven planning of product-line evolution. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, FOSD '09, pages 109–116, New York, NY, USA, 2009. ACM.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In R. Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer Berlin Heidelberg, 2004.

[8] K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*, pages 16–20, 2005.

[9] N. Dintzner, U. Kulesza, A. van Deursen, and M. Pinzger. Evaluating feature change impact on multi-product line configurations using partial information. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, volume 8919 of *Lecture Notes in Computer Science*, pages 1–16. Springer International Publishing, 2014.

[10] N. Gamez and L. Fuentes. Software product line evolution with cardinality-based feature models. In *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse*, ICSR'11, pages 102–118, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] W. Heider, R. Rabiser, P. Grünbacher, and D. Lettner. Using regression testing to analyze the impact of changes to variability models on products. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 196–205, New York, NY, USA, 2012. ACM.

[12] K. Kang. *Feature-oriented Domain Analysis (FODA): Feasibility Study ; Technical Report CMU/SEI-90-TR-21 - ESD-90-TR-222*. Software Engineering Inst., Carnegie Mellon Univ., 1990.

[13] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the safe evolution of software product lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 33–42, New York, NY, USA, 2011. ACM.

[14] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba. Coevolution of variability models and related artifacts: A case study from the linux kernel. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 91–100. ACM, 2013.

[15] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[16] C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, and G. Botterweck. Consistency checking for the evolution of cardinality-based feature models. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 122–131, New York, NY, USA, 2014. ACM.

[17] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, and U. Kelter. Fault-based product-line testing: effective sample generation based on feature-diagram mutation. In *Proceedings of the 19th International Conference on Software Product Line*, pages 131–140. ACM, 2015.

[18] C. Seidl, I. Schaefer, and U. Aßmann. Capturing variability in space and time with hyper feature models. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 6:1–6:8, New York, NY, USA, 2013. ACM.

[19] C. Seidl, I. Schaefer, and U. Aßmann. Integrated management of variability in space and time in software families. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 22–31, New York, NY, USA, 2014. ACM.

[20] T. Thüm, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society.

[21] J. White, J. A. Galindo, T. Saxena, B. Dougherty, D. Benavides, and D. C. Schmidt. Evolving feature model configurations in software product lines. *J. Syst. Softw.*, 87:119–136, Jan. 2014.