Model-driven support for product line evolution on feature level

Andreas Pleuss a,∗, Goetz Botterweck a, Deepak Dhungana b, Andreas Polzer c, Stefan Kowalewski c

a Lero, University of Limerick, Ireland
b Siemens AG Österreich, Vienna, Austria
c RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany

A B S T R A C T

Software Product Lines (SPL) are an engineering technique to efficiently derive a set of similar products from a set of shared assets. In particular in conjunction with model-driven engineering, SPL engineering promises high productivity benefits. There is however, a lack of support for systematic management of SPL evolution, which is an important success factor as a product line often represents a long term investment. In this article, we present a model-driven approach for managing SPL evolution on feature level. To reduce complexity we use model fragments to cluster related elements. The relationships between these fragments are specified using feature model concepts itself leading to a specific kind of feature model called EvoFM. A configuration of EvoFM represents an evolution step and can be transformed to a concrete instance of the product line (i.e., a feature model for the corresponding point in time). Similarly, automatic transformations allow the derivation of an EvoFM from a given set of feature models. This enables retrospective analysis of historic evolution and serves as a starting point for introduction of EvoFM, e.g., to plan future evolution steps.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

The core idea in product line engineering (PLE) is to invest into the development of a product line (Software Product Line, SPL), such that individual products can later be derived more efficiently. Concepts from model-driven engineering (MDE) can be used to further increase efficiency and automation. The variability among different products is often represented in models, e.g., by feature models (Kang et al., 1990), which describe available features and dependencies between them in hierarchical structures. A concrete product is then defined by a configuration of this model (i.e., selected and eliminated features).

A product line is a long-living entity and it must evolve to meet the new requirements over many years. It is therefore impossible to foresee all changes and scope the SPL accordingly right from the beginning. Changes can, e.g., be introduced by new technologies, changed customer preferences, or competitors. There is a general consensus that in the context of product line engineering (PLE) evolution is an important aspect (Bosch, 2002; Chen, 2004; Schach and Tomer, 2000). Nevertheless, the strategic management of product line evolution has not been covered by the literature extensively (see Section 2). Instead, most researchers report on approaches to support changes in the implementation, while the evolution is “handled on the fly”, i.e., the product line is extended from case-to-case to satisfy the requirements of new products.

Fig. 1 shows a small example feature model for a SPL in the automotive domain. It shows different versions of the same model at different points in time. The individual models use a FODA-based notation (Kang et al., 1990). In 2009, radio is an optional feature. In 2010, navigation is added, which requires the radio for its user interface as it shares the same display. Due to a management decision, navigation is separated from the radio in 2011 having a display of its own. In 2012, navigation is enhanced with data storage.

Such changes could be handled spontaneously, but to ensure sustainable success one has to follow a systematic approach which makes evolution explicit. In the context of model-driven PLE, this should be done by a model-driven approach, which integrates with the existing modeling concepts and allows the specification of evolution on an appropriate abstraction level.

Hence, we present a model-driven approach to handle the evolution of SPLs on feature model level. Our motivation is to reap the benefits of model-driven techniques (e.g., documentation, automation, foundation for customized visual representations) for SPL evolution. Our work is geared towards the following goals:

G1 Documentation (of previous evolution): Disclose hidden knowledge by explicitly specifying evolution in a structured way including associated rationale.
Planning (of future evolution): Documentation should not only include previous evolution but also support to make statements about future versions.

Abstraction: Reduce cognitive complexity of feature models by additionally supporting abstraction from single changes by subsuming multiple related changes.

Automation: Increase efficiency and reduce errors by providing automated tool support for the transition between the evolution model and the feature models themselves.

Analysis: Reduce errors by detecting inconsistencies or potentially contradictory or incomplete changes.

The remainder of the paper is structured as follows: We first discuss related work and its shortcomings in Section 2. We provide an overview of our EvoPL framework in Section 3. Subsequently, the various models are described in more detail in Section 4. We show the model transformations, additional tool support, and the analysis with EvoPL in Section 5. Finally, we report on an initial evaluation of the presented concepts Section 6 and conclude the paper with final thoughts (Section 7).

2. Related work

In this section we will provide an overview of related work on SPL evolution in general. Then, we focus on existing concepts, which we could use to model evolutionary changes between feature models. Finally, we discuss shortcomings of these approaches.

2.1. Evolution of product lines

PLE should treat evolution as the normal case and not as the exception (Dhungania et al., 2008). Despite this importance, surprisingly few approaches deal with product line evolution (e.g., Bosch, 2000; McGregor, 2003; Svahnberg and Bosch, 1999). Evolution support is of particular importance in model-driven PLE, e.g., to ensure consistency after changes. Several authors (Johnson and Bosch, 2000; Siy and Perry, 1998) stress the importance of approaches for product line evolution to avoid the erosion of a product line. Some existing work deals with implementation issues for evolving product lines (Loughran and Rashid, 2004; Deng et al., 2005) using, e.g., aspect-oriented programming. Mende et al. (2008) describe tool-support for the evolution of product lines based on the “grow-and-prune” model, i.e., they support refactoring code that has been created by copy & paste and which can potentially be propagated to product line level. Svahnberg and Bosch (1999) report on experiences regarding the evolution of products, components, and software architecture.

Deelstra et al. (2005) focus on product derivation. Besides other aspects they discuss types of SPL adaptations (product-specific, reactive, proactive) and the scope of adjustment. According to this taxonomy, our approach addresses proactive evolution, i.e., active planning of future versions on domain level.

2.2. Modeling changes on models

As feature model changes are in the focus of this paper, we discuss existing concepts for modeling differences between models. We classify and describe them in three categories, i.e., Model Comparison, Delta Models, and Change Operators. Subsequently, we discuss how those concepts need to be extended to support proactive planning of feature model evolution.

Model Comparison. When dealing with evolution of feature models, the easiest way to capture commonalities and differences between different evolution steps is usage of general model comparison tools like EMFCompare (Eclipse-Foundation, 2011c) or Epsilon ECL (Eclipse-Foundation, 2011d). These tools apply concepts from text-based comparison tools to models. The differences between two models can be visualized and stored as a “Diff Model” for further processing. However, these tools are not sufficient for model evolution as they do not provide much abstraction or support for planning. Also, they are intended for comparing a small number of models only, which is not sufficient when dealing with long-term evolution.
Delta Models. A more powerful technique to specify differences between models are model fragments and Delta Models (also called change sets). Model fragments (or “model snippets”) (Ramos et al., 2007) can be used in model-driven PLE to implement “positive variability”, i.e., the composition of products according to configurations (Voelter and Groher, 2007). The composition can then be achieved using techniques for model merging (Eclipse-Foundation, 2011d) or model weaving (Eclipse-Foundation, 2011a). Dhungana et al. (2008) use fragments of variability models to support parallel evolution by multiple teams.

Delta Models can be used to specify changes on a given model. Schaefer (2010) uses Delta Models for incremental model refinement and provides a formalization of the Delta Models. Hendrickson and van der Hoek (2007) describe a tool for specifying Delta Models (which they call “change sets”) and relationships between them to describe architectural variability.

Change Operators. An alternative concept to deal with model changes are Change Operators. As shown by Rose et al. (2009), this is the most common way used in “model co-evolution” to keep models synchronized. The three atomic kinds of model changes are to add, delete, or modify elements. More complex Change Operators (i.e., combinations of atomic operators) allow semantically rich descriptions of changes (e.g., “insert subclass”). Some authors (Lerner, 2000; Wachsmuth, 2007; Herrmannsdörfer et al., 2009) provide sets of Change Operators specific for metamodels.

Some authors specifically address feature models. Thüm et al. (2009) present a tool to classify whether changes result in a specialization (enlarging the set of products), a generalization (reducing the set of products), a refactoring of the model, or an arbitrary change. Other work provides Change Operators for specific tasks on feature models, e.g., generalization (Alves et al., 2006) or specialization (Czarnecki et al., 2005). Follow-up work by Kim and Czarnecki (2005) also discusses feature model changes for co-evolution, including addition and deletion of nodes, moving nodes or subtrees, and changes of cardinalities. As we discussed in Botterweck et al. (2010), existing sets of Change Operators are not sufficient for proactive SPL evolution. Thus, we presented a catalogue of evolution operators (Botterweck et al., 2010) which is used in the approach presented here (see Section 4.3). Further technical details of our evolution framework are described in Pleuss et al. (2010).

2.3. Shortcomings of existing modeling concepts

This section discusses the coverage and shortcomings of existing modeling concepts for our purpose. We first show the requirements resulting from the five goals (G1–G5) introduced in Section 1 and then discuss existing modeling concepts with respect to them.

Our first goal, documentation of previous evolution (G1), requires concepts to represent changes, associated rationale, and a notion of time. To support long-term planning of future evolution (G2) it is also desirable to allow order-independent specification of models. For instance, in practice, one might want to make statements about the version in 2016 (e.g., “in 2016 we will support head-up displays”) even if the version in 2015 is not fully specified yet. This includes the need to explicitly specify changes as undecided yet (e.g., whether “head-up display” will be supported in 2015 already or not) to indicate that they must be decided in the future. The goal of abstraction (G3) requires new modeling concepts, to encapsulate the fine grained evolution steps into larger conceptual changes. The goal of analysis (G4) requires concepts for the definition of dependencies between changes, e.g., that one feature is replaced by an alternative, which implies that they cannot both be present at the same time. From the goal of automation (G5) we can derive that the modeling concepts require support for automated processing, e.g., by providing a clearly defined language with precise semantics.

Table 1 summarizes these requirements and existing support in related work. We distinguish here between two general categories, Delta Models and Change Operators. Both of them support, of course, specification of changes between models. Rationale and a notion of time is usually not part of these approaches but they can be extended with such concepts.

A shortcoming of existing approaches is their limited support for order-independent planning and undecided decisions. Both Delta Models and Change Operators specify changes relative to a fixed reference point. We illustrate this in Fig. 2 using Delta Models (Hendrickson and van der Hoek, 2007). There are two alternatives for a reference point: The first option is that changes refer to the previous version (Fig. 2(a)). At first look specifying changes relative to the previous version seems a natural way for expressing evolutionary changes. However, it does not allow order-independent long-term planning; for instance, specifying decisions about the version in 2015 is not possible before 2014 has been specified. The second alternative is that changes refer to a baseline (Fig. 2(b)). This allows order-independent specification of versions. The drawback, however, is redundancy because remaining changes have to be repeated over and over again. Also, this variant makes it harder to see differences between versions. In summary, both alternatives lack support for order-independent planning.

Finally, dependencies between changes are supported by Delta Models (Hendrickson and van der Hoek, 2007). Other approaches could not be extended with this concept. Both, Delta Models and Change Operators usually have precise semantics defined and can potentially be used in automated approaches.

In the next section we present our EvoPL approach, which combines existing and new concepts to address the shortcomings identified here.

3. The EvoPL framework

This section introduces the EvoPL (Evolving Product Line) framework, a model-driven approach to plan and manage product line evolution on feature level. EvoPL consists of the following elements:

(1) To gain abstraction, we use Fragments to cluster feature model elements that are added or removed during the same evolution step. Fig. 3(a) shows fragments corresponding to the feature models in Fig. 1; each one is represented by a name. Each feature model (at a certain point in time) can be described as a composition of fragments. Fragments are described in Section 4.1.

(2) To specify changes within fragments we use Change Operators (Botterweck et al., 2010), called EvoOperators in the context of EvoPL. While fragments are used to specify which elements are
present in the feature model, EvoOperators are used to specify modifications on those elements, e.g., turning a mandatory feature into an optional one or moving features within the model. For instance in Fig. 1, Navigation requires Radio in 2010 but not in 2011. This is specified by an EvoOperator requires radio, which is applied in 2010 but not in 2011. EvoOperators are described in Section 4.3.

(3) To specify relationships between the fragments (i.e., the hierarchical structure and other dependencies), we reuse concepts of feature models. We call this special kind of feature model Evolution Feature Model (EvoFM). Fig. 3(b) shows the EvoFM for the example, preserving the hierarchical structure of the fragments. The EvoOperators are part of the EvoFM as well (marked by angle brackets). For instance, requires radio is added to the fragment Navigation. A configuration of the EvoFM, called EvoConfiguration, specifies which fragments and EvoOperators are selected for an evolution step. In this way, the complete feature model can be derived from an EvoConfiguration by composing EvoFragments and applying EvoOperators. Each EvoConfiguration is associated with a point on the global Timeline. Configuration decisions can be associated with Rationale. The EvoFM and EvoConfigurations are described in Section 4.2.

(4) The modeling concepts above can be represented by a visualization, which we call Evolution Plan. An Evolution Plan represents all EvoConfigurations over time, similar to a product matrix in a product line, but representing evolution steps instead of products. Fig. 4 shows the Evolution Plan for the example from Fig. 1. The horizontal dimension represents the timeline; each column represents an evolution step. The vertical dimension represents the EvoFM; each row represents an EvoFM element, e.g., a fragment or an EvoOperator. Each cell represents an

Fig. 2. Alternative reference points by the example of Delta Models. (a) Relative to previous model. (b) Relative to baseline.

Fig. 3. The EvoFM for the feature models in Fig. 1. (a) Clustering into fragments. (b) Corresponding EvoFM.

Fig. 4. Evolution Plan for the example in Fig. 1.

EvoConfiguration decision, i.e., whether the EvoFM element is active in that version.

The Evolution Plan provides the central view when working with EvoPL. It has to be complemented with two additional views showing (a) the content of fragments and (b) the rationale associated with EvoConfiguration decisions. The Evolution Plan can be used to review, plan, and specify product line evolution on feature level by adding new configurations and new fragments and EvoOperators. Automated model transformations allow to generate an Evolution Plan from a given sequence of feature models and to generate a feature model from the Evolution Plan. In the remainder of the paper we will first describe the underlying modeling concepts (Section 4) and then the process of using these concepts Section 5.

4. EvoPL models

Here we explain the models of the EvoPL framework in more detail. Their abstract syntax is defined by a metamodel, which is implemented using the Eclipse Modeling Framework (EMF) (Eclipse Foundation, 2011b).

In our approach we use feature models based on FODA (Kang et al., 1990). For the purpose of this article we use the metamodel shown in Fig. 5, which includes basic elements like Feature and FeatureGroup (generalized as FMNode) and dependencies (ExcludesDependency, RequiresDependency). If desired this can be extended by advanced concepts (e.g., feature cardinalities and other constraints) and this would not limit the applicability of our approach. Moreover, we provide model transformations to import other formats, e.g., from AHEAD (Batory, 2005) and our own S2T2 Configurator (Botterweck et al., 2009).

Here, we focus on evolution of product lines as described by feature models. Further product line models (e.g., implementation models) are beyond the scope of this paper. However, we briefly discuss the propagation of evolution changes (Section 7).

The following subsections describe the modeling concepts introduced earlier in more detail, i.e., the EvoFragments (Section 4.1), the EvoFM (Section 4.2), the EvoOperators (Section 4.3), and finally the Rationale to be associated with configuration decisions (Section 4.4).

4.1. EvoFragment

An EvoFragment defines a feature model subtree that is added or removed coherently during evolution. Fig. 6 shows the metamodel for fragments. Each fragment has a ContextRootNode defining its position in the overall feature model. The context root node “clones” a node in another EvoFragment which serves as root node for the fragment (“node” here refers to FMNode from Fig. 5, i.e., a feature or a feature group). The content of the EvoFragment is specified as children of ContextRootNode.

Fig. 7 shows several fragments from the example in Fig. 3(a), here with their context root nodes displayed (in white color). The fragment CarBody does not have a context root node as it is the root fragment of the feature model (Fig. 7(a)). The fragment Navigation contains only a single feature Navigation with MultimediaDevices as context root node (b). The fragment NavigationDisplay is an example with a feature group (c).
within different fragments, then an additional context node is used representing the target feature, like DVDDrive (e). In the metamodel in Fig. 6, a special subclass DependencyContext is defined for this purpose to be used as target for the dependency. In addition, it must be ensured that this fragment is used always in conjunction with the fragment containing the dependency context. Such constraints are defined using the dependencies in the EvoFM, as shown in the next section.

4.2. EvoFM

To specify the dependencies and relationships between the EvoFragments we use feature model concepts itself specified in a special kind of feature model called EvoFM. As shown in its metamodel (Fig. 8) an EvoFM is structured analogously to feature models (cf. the metamodel in Fig. 5). However, instead of features it contains EvoFragments (see previous section) and EvoOperators (see next section).

The parent–child relationships in the EvoFM are used to specify the hierarchy of EvoFragments. An EvoFragment \( f \) is child of the EvoFragment which contains its context root node. Using the conventional semantics of parent–child relationships in feature models (selection of a child requires selection of a parent) it is always ensured that all context root nodes are available. The EvoFragment which contains the root node of the (product line) feature model has no context root node and is the root of the EvoFM (CarBody in the example).

Additional dependencies between fragments are specified using the conventional cross-tree dependencies of feature models. An example are dependencies in the (product line) feature model spanning multiple fragments such as in Fig. 7(e). In this case, a requires dependency is defined between the EvoFragments as shown in Fig. 3.

A concrete feature model on product line level is defined by a configuration (i.e., selection of EvoFragments and EvoOperators) of the EvoFM, called EvoConfiguration. The metamodel for EvoConfigurations is shown in Fig. 9. Each EvoConfiguration is associated with a point in time from the global timeline. A simple metamodel for the timeline is shown in Fig. 10. The concept PointInTime can be further refined into dates, version names, or any other concept relevant for the evolution.

An EvoConfiguration consists of multiple configuration decisions (here EvoDecision) selecting and deselecting the EvoFM elements (i.e. Fragments and EvoOperators). For the purpose of planning it is important to support the value undecided (which is

Fig. 7. Examples of EvoFragments with context nodes. (a) CarBody. (b) Navigation. (c) Navigation Display. (d) Color Radio Display. (e) DVD Entertainment. (f) Data Storage.

Fig. 8. Metamodel for EvoFM models.
the default value) to explicitly state that a decision has not been made yet.

Selecting an EvoFragment means that the content of the fragment is part of the feature model for this point in time. Deselecting a fragment means that the fragment’s content is not present in the feature model. The content of fragments is disjoint and the union of all fragments specifies all available feature model elements from which the different versions of the feature model are composed. Specifying new feature model elements means creating a new fragment in EvoPL. To specify evolutionary changes on the elements within an EvoFragment, the EvoOperators must be used (see next section). Thus, EvoFragments usually do not need to be modified themselves when working with EvoPL. On exception is the splitting of a fragment, when only a part of it should be added to a feature model version. Splitting is further discussed in Section 5.1.

4.3. EvoOperators

As explained above, feature models are specified by EvoConfigurations. This means that adding or removing elements from a certain feature model version is defined by selecting and deselecting EvoFragments. Any other modifications (beside “add” and “remove”) are specified by EvoOperators. An EvoOperator specifies the change of one or more properties of feature model elements. This includes moving features in the model, changing the feature type from optional to mandatory, grouping features or adding a dependency.

For instance, in Fig. 1, Navigation has a requires relationship to Radio because it uses the same display. Thus, this requires relationship would be part of the fragment containing Navigation. However, in 2011, Navigation has its own display, so the requires relationship is no longer necessary. Thus, the relationship is represented by an EvoOperator requires Radio; which is selected in 2010 and deselected in 2011 (see Fig. 4).

Beside modification of single properties, we also support EvoOperators with richer semantics, like inserting a feature group or moving all subfeatures of a feature. In Botterweck et al. (2010) we presented a catalogue of EvoOperators for feature models. Table 2 shows the (slightly revised) syntax for all supported EvoOperators.

In the EvoFM, an EvoOperator is a child of the EvoFragment containing the elements modified by the operator. An EvoOperator is a specific type of an EvoFeature and can thus be selected or deselected as part of an EvoFM configuration and is visible in the evolution plan. All EvoOperators refer to feature model elements defined in the EvoFragments and never to the result of other EvoOperators. They are thus order independent. Selecting an EvoOperator in an EvoConfiguration means that the operator is applied to this feature model. It is possible that EvoOperators conflict each other when they refer to the same model elements. To avoid conflicts one can specify cross-tree constraints between EvoOperators like in conventional feature models, for instance, an XOR feature group or an excludes relationship.

4.4. Rationale

To document reasons and background info leading to a decision, each evolution decision can be augmented with a rationale. For instance, in the example in Fig. 1, in 2010 the requires relationship between Navigation and Radio is introduced as Navigation shares the display with the Radio. This should be documented with a short description like “Navigation shares display with Radio”.

Fig. 11 shows the metamodel for rationale descriptions. A Rationale contains a textual description and can be assigned to one or more decisions. Assigning the same rationale to multiple decisions is useful if multiple decisions (within the same version, over different versions, or both) are made for the same reason. In turn, a single decision can also be associated with multiple rationale descriptions. For instance in Fig. 1, DVD Drive is introduced to support DVD entertainment but also as a data storage for additional maps.

Often, there are dependencies between rationales. For instance, a rationale can be subordinated to another one like the rationale “Navigation shares display with Radio” which contributes to the introduction of the navigation system as a whole. Analogously, two rationale can contradict each other, e.g., if they are associated with mutually exclusive decisions. Often such dependencies are also reflected in the feature model itself (e.g., by the parent–child relationships) but it should still be possible to specify the dependencies on rationale level, too. To this end, we support ContributesTo and Contradicts relationships between rationales.

As the discussion shows there is potential for further work in this direction. For instance, evolution planning could involve specification of high level goals (or a hierarchy of goals), assigning rationale to goals, and specification of various dependencies between them similar like in goal-driven requirement engineering approaches like KAOS (Van Lamsweerde, 2001). However, the temporal dimension adds further complexity as goals themselves can change over the time as well. This area needs further investigation and is not discussed in more detail in this article.

5. Working with EvoPL

In this section we will explain how the EvoPL concepts described in the preceding section can be implemented and applied such that the goals set in the introduction (e.g., modeling, planning, analyzing evolution) can be achieved.

Fig. 12 shows an overview of the workflow. First, we provide a model transformation that extracts an EvoFM from a sequence of existing feature models. We describe this in Section 5.1. The extracted EvoFM is then refined by the developer and used to plan future evolution (Section 5.2). Subsequently, the specified evolution can be analyzed and refined (Section 5.3). Once the planning of the next version is finished, the resulting feature model is generated by a model transformation (Section 5.4).
5.1. Extraction of EvoFM from existing FMs

This section describes how an EvoFM and EvoConfigurations can be derived from given feature models. To do this we use an incremental approach: The input is an existing EvoFM (with EvoConfigurations) and a single Feature Model (FM) for an additional evolution step. The output is an updated EvoFM and EvoConfigurations, which include the additional information from the input FM (Fig. 13). In the initial situation, when no EvoFM exists, the output is a trivial EvoFM, which contains only a single EvoFragment which contains the whole FM. In subsequent steps the model transformation adds EvoFragments for new feature model elements, splits EvoFragments when required, and adds EvoOperators for modifications of elements in EvoFragments.

**Table 2**
Syntax and informal semantics of supported EvoOperators.

<table>
<thead>
<tr>
<th>Change Operators for Features</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>move to g</td>
<td>Moves the feature below g</td>
</tr>
<tr>
<td>to {mandatory</td>
<td>optional} [move to g]</td>
</tr>
<tr>
<td>subfeatures [f1, f2, ..., fn] move to g</td>
<td>Moves all subfeatures (or optionally some selected subfeatures f1, f2, ..., fn) below g</td>
</tr>
<tr>
<td>subfeatures [f1, f2, ..., fn] to {mandatory</td>
<td>optional} [move to g]</td>
</tr>
<tr>
<td>subfeatures [f1, f2, ..., fn] to {or group</td>
<td>xor group} [move to g]</td>
</tr>
<tr>
<td>requires g</td>
<td>Inserts “requires” dependency to g</td>
</tr>
<tr>
<td>excludes g</td>
<td>Insert “excludes” dependency to g</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change Operators for Feature Groups</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>move to g</td>
<td>Moves the feature group below g</td>
</tr>
<tr>
<td>subfeatures to {or group</td>
<td>xor group} [move to g]</td>
</tr>
<tr>
<td>subfeatures [f1, f2, ..., fn] to {or group</td>
<td>xor group} [move to g]</td>
</tr>
<tr>
<td>subfeatures to {mandatory</td>
<td>optional} [move to g]</td>
</tr>
<tr>
<td>subfeatures [f1, f2, ..., fn] to {mandatory</td>
<td>optional} [move to g]</td>
</tr>
</tbody>
</table>

Legend: purple=EvoFM keywords, blue-features

**Fig. 11.** Metamodel for the rationale descriptions.

**Fig. 12.** The workflow with EvoFM.
Let us consider the situation from Fig. 1 and assume that the FM are given and no EvoFM exists yet. Our transformation takes the FM for 2009 and, as no EvoFM exists, creates a trivial EvoFM with a single EvoFragment (by default named as its root node, here CarBody) containing the FM for 2009 and an EvoConfiguration for 2009 where this single EvoFragment (CarBody) is selected. In the second step, this EvoFM is taken as input together with the next FM for 2010. In the example, only one EvoFragment has to be added for 2010, containing the feature Navigation and its requires relationship. When adding the FM for 2011 to the resulting EvoFM, an additional EvoFragment is added for the navigation displays. In addition, the EvoFragment CarBody has to be split since ColorRadioDisplay is not part of the FM in 2011 while all other elements from the fragment CarBody are. Thus, ColorRadioDisplay has to be split into a separate EvoFragment. Moreover, an EvoOperator has to be added as the requires relationship is removed from the fragment Navigation. The last transformation step for 2012 results in the EvoFM and EvoConfigurations as shown in Figs. 3 and 4.

The transformation rules can be summarized as follows: First, a preceding “match phase” is performed, where the features in the FM are compared with features in existing fragments. The result of the comparison is stored so that each feature is marked as “matched” if it has a counterpart or “unmatched” otherwise. Then, the following transformation rules are applied:

1. A new EvoConfiguration is created for the source FM.
2. Each EvoFragment is copied to the target EvoFM. If all its contents (children of the context root node) are marked as “unmatched” then it considered as deselected in the new EvoConfiguration. If all its content is marked as “matched”, then it is set to selected. If some content is “matched” and some “unmatched”, the fragment is split into a EvoFragment containing the “matched” elements (selected in the new EvoConfiguration) and an EvoFragment containing the “unmatched” elements (deselected in the new EvoConfiguration). In addition, all existing EvoConfigurations are updated by replacing the old EvoFragment by the new ones resulting from the split.
3. Each “matched” element in an EvoFragment has to be compared to its counterpart in the FM regarding differences in its properties (like mandatory or optional). For each difference an EvoOperator is created.
4. For “unmatched” elements in the FM a new EvoFragment has to be created, which is set as selected in the EvoConfiguration. All unmatched elements in the same subtree (i.e. related by parent–child relationships) are put into the same new EvoFragment.
5. Other elements from the source EvoFM (like existing EvoConfigurations) are copied into the target EvoFM.

In a subsequent phase, new EvoOperators are checked to determine whether they can be combined into more complex EvoOperators.

The model transformation has been implemented using the Epsilon Transformation Language (ETL). Names of created EvoFragments are by default taken from the root elements in the fragment. If multiple root nodes exist (like NavigationDisplay in the example) the name is composed from the names of the children. In some cases this does not result in sufficiently expressive names. Then, the developers sometimes have to edit the generated name if desired.

5.2 Planning future FM evolution

This section describes how to plan and specify evolution with EvoFM (3 in Fig. 12). We first discuss the manual refinements, which might become necessary when using an automatically extracted EvoFM. Then, we specify the typical planning steps. Finally, we discuss how interactive tools can provide useful additional support.

5.2.1. Refinement of a generated EvoFM

If the EvoFM was extracted automatically using our model transformation it is first useful to refine the generated model. For example, automatically generated names of EvoFragments can be replaced with more meaningful labels based on domain knowledge. Apart from that, rationale descriptions can be associated to the evolution decisions. As described in Section 4.4, a rationale can be assigned to multiple decisions and it is also possible to specify dependencies between them.

5.2.2. Planning with EvoFM

The Evolution Plan can then be used to guide future evolution. This is done by defining a new point in time (i.e., EvoConfiguration) and specifying the evolution decisions. One can specify new
points in time and decisions in arbitrary order, thanks to the order-independent approach of EvoFM (see Section 2.3). Again, rationale documentation can be attached to decisions.

If new elements are added to the EvoFM, a new EvoFeature (incl. fragment) has to be created to which new FM elements can be added. Even if the concrete FM elements to be added are not known yet, an EvoFeature with a meaningful name and associated rationale can be used to document the intent. The second operation in EvoFM is to split existing fragments (cf. Section 4.1). This is necessary if only a subset of elements from an existing fragment should be included in a new FM version.

As described in Section 5.1, the incremental nature of the transformation allows to automatically extract information from additional feature models to the EvoFM at any time. This means, that during planning with EvoFM it is still possible to add information from a feature model version which has not been considered yet or to perform changes on a feature model directly and add this information to the evolution plan.

5.2.3. Interactive tool support

EvoPL is intended to be supported by interactive visual tools. Fig. 14 shows a user interface prototype. It consists of three main views: The Evolution Plan, the Fragment View which shows the content of an EvoFragment from the Evolution Plan, and the Feature Model View, which shows the result of a EvoConfiguration. This has to be complemented with a view showing the rationale documentation associated with EvoConfiguration decisions. For the EvoFM itself (leftmost column in the Evolution Plan), additional interactive techniques should be supported like for conventional feature models (Botterweck et al., 2008).

The tool can support operations on EvoFM, e.g., splitting of fragments such that the user only has to select elements (in the fragment where the split should occur) and give a meaningful name to the new EvoFragment while the EvoConfigurations are updated automatically. The tool also notifies the user if EvoFM constraints are violated, e.g., when an EvoFeature is selected that depends on another EvoFeature that has not been selected. Since the underlying semantics can be mapped to propositional logic, this can be implemented with a SAT solver similar to our S2T2 Configurator for feature models (Botterweck et al., 2009). In addition, the results of analysis (see next section) are displayed as warnings, similar to, e.g., a “problems” view in IDEs like Eclipse.

5.3. Analysis of evolution

EvoPL framework allows to perform a model-based analysis of the historical and planned evolution. This section discusses different aspects to be analyzed: the structural consistency of models, consistency of an EvoConfiguration, and a semantic analysis of the evolution itself based on the relationships between rationale descriptions.

5.3.1. Structural consistency

The structural consistency of a model can be analyzed without deep knowledge of the semantics. Examples for potential inconsistencies are broken references, identifiers which are not unique, or elements without required child elements. Most of the underlying structural constraints are defined directly by the metamodel, e.g., by cardinalities in the metamodel (see Section 4). More complex constraints can be attached to metamodel elements, e.g., as OCL constraints, for instance to ensure that the context root node in an EvoFragment is a feature or feature group also found in the parent EvoFragment (see Section 4.2).

Modeling frameworks like EMF Eclipse-Foundation (2011b) ensure that models comply to the metamodels, manage the references between elements, and allow to validate additional constraints. As EvoPL is defined as a single, consistent metamodel implemented with EMF, structural consistency can be ensured without the need for custom analysis tools.

5.3.2. Consistency of EvoConfigurations

Since in EvoFM the FMs are specified by EvoConfigurations (Section 4.2), the EvoConfigurations must be constrained so that they lead to (structurally) correct feature models. As EvoFM is a specific kind of feature model, we reuse the existing feature model semantics (Schobbens et al., 2006) to define constraints for EvoConfigurations. The parent–child relationships in the EvoFM ensure the correct hierarchy of EvoFragments as, by definition (see Section 4.2), each context root node in an EvoFragment is an element from its parent EvoFragment (and the parent–child relationship in feature models requires that for each selected child the parent is selected as well). Additionally, there can be further custom dependencies defined in the EvoFM by using feature groups, cross-tree constraints or specifying EvoFragments as mandatory.

All these feature model constraints can be checked using existing tools for feature configuration. For instance our S2T2 feature configurator tool described in Botterweck et al. (2009) automatically translates feature models into propositional logic uses a SAT solver derive consequences of decisions. It ensures that remaining configurations always fulfill constraints set out in the model.

Some other constraints are specific to EvoFM: the relationships between EvoFragments defined by dependency context and the EvoOperators. If an EvoFragment f is selected which contains a dependency context (Section 4.1) referring to another EvoFragment g, then g has to be selected as well. This can be ensured by specifying a requires constraint between f and g as soon as the dependency context is specified (e.g., manually or by a tool).

Two constraints have to be considered for the EvoOperators: First, all elements referenced by a selected EvoOperator must be available, i.e. the EvoFragment they are contained in has to be selected. For the elements to be manipulated by the EvoOperator this is already ensured as, by definition, an EvoOperator is specified in EvoFM as a child of the EvoFragment which contains the elements to be manipulated (Section 4.3). However, the target elements of a moveTo operation can be part of another fragment. Again, a requires constraint can be specified in EvoFM to ensure this constraint.

Second, conflicts between EvoOperators can occur if multiple EvoOperators influence the same element (e.g., when two moveTo operators try to move the same feature). To avoid this it is necessary to identify operators that target the same element. The user can then specify additional constraints in the EvoFM to avoid conflicts between EvoOperators in advance, like specifying EvoOperators as an xor-group or by specifying an excludes relationship between them.

5.3.3. Semantic analysis of evolution

In addition, it is also possible to provide a semantic analysis based on the rationale descriptions. This helps to identify candidates for unintended changes beyond model inconsistencies, like potentially incomplete changes or contradictory changes. As introduced in Section 4.4, a rationale can be associated with multiple changes and Conflicts and contributesTo relationships can be defined between the rationale. Cases for potential unintended changes are:

- if multiple changes are introduced based on the same rationale and in a later version only a subset of them is taken back,
- if multiple changes are introduced whose rationale are related by a contributesTo relationship and in a later version only a subset of them is taken back, and

• if multiple changes whose associated rationale are related by a Conflicts relationship are selected in the same version.

In addition, it might be possible to identify evolution patterns based on the evolution plan and to provide recommendations based on them. For instance, if multiple related changes (e.g., related by parent–child relationships or rationale) are planned within multiple consecutive versions, it might (depending on the application domain) be more efficient to introduce them all within a single step instead (e.g., to minimize the number of changes on a subsystem) or vice versa (e.g., to prefer gradual changes over too large ones). Although, the identification of such patterns requires experience and domain knowledge, the concepts and mechanisms provided by EvoFM can help to handle such situations more efficiently.

5.4. Generation of resulting feature models

From the EvoFM and a given EvoConfiguration it is possible to automatically derive the corresponding FM. This requires that the EvoConfiguration no longer contains undecided decisions. Then, the selected fragments have to be composed using their context root nodes. This is a straightforward model transformation which directly creates the corresponding feature models. The creation of FM from an EvoFM and an EvoConfiguration has been implemented as a model transformation with ETL.

It is also possible to propagate the FM changes to more concrete models, such as a component model associated with the FM. We have performed first experiments into this direction using an embedded systems product line implemented with Matlab/Simulink as example. In this product line (see Polzer et al., 2009 for details), each feature is represented by a special Simulink subsystem called Module. Variability is realized by a negative variability approach, i.e., the product derivation starts with a superimposed model containing all existing Modules from which those Modules are removed which are associated with deselected features (Botterweck et al., 2009).

The first step in propagating changes to the Simulink model is to automatically generate a placeholder Module (and a link to the corresponding feature) for each feature which has been added during evolution. In turn, Modules corresponding to deleted features during an evolution step can be removed using the same negative variability mechanism as for product derivation.

6. Evaluation

In this section we evaluate the usefulness of the presented approach by applying it to data from a large-size real-world project. The question we aim to answer is: Are the EvoFM modeling concepts suitable to model large real-world feature models in an efficient way? We first describe the study and its results in Section 6.1 and then discuss threats to validity in Section 6.2.

6.1. Application to real data

For our study we use the open-source platform Eclipse.1 This is not a SPL in strict sense but fulfills many SPL characteristics (Bermejo and Dai, 2006; Ahmed et al., 2008). We chose Eclipse as it is open-source, large, evolving, and provides a high degree of well-defined variability which can be interpreted in terms of features. Eclipse consists of a small kernel and well-defined extension points. Extensions are provided in form of Eclipse Plugins which are developed by the Eclipse project and its sub-projects as well as by third-party tool vendors.2 The functionality is organized as Eclipse Features, which contain Plugins and may require other features and plugins.

To extract data about the different versions we used the Eclipse update site mechanism. We developed a small script to mirror a sequence of update sites and implemented a tool, which then

---

2 http://www.eclipse.org/projects/.
extracts the information about Features and Plugins, e.g., from the feature.xml and plugin.xml files. The extracted information is then stored as a model and transformed into feature models, which conform to our metamodel (cf. Section 4). Each Eclipse Feature and Eclipse Plugin is interpreted as a SPL feature in the feature model. The interpretation follows these rules: An Eclipse Plugin p contained in an Eclipse Feature f is interpreted as mandatory child feature of f. An Eclipse Feature f including another Eclipse Feature g (i.e., extending its functionality) is interpreted as an optional child of g (as in feature models children require their parents). If an Eclipse Feature f includes multiple other Eclipse Features g_i (like a whole SDK, which includes multiple features), it is interpreted as an (optional) sibling of g_i with requires relationships to all of them. Eclipse Features specify also requires relationships between Eclipse Features and between Plugins which are mapped to requires relationships in the feature model.

For our study we used the data from the Eclipse Platform project, which provides the core frameworks and services for Eclipse. We considered the annual minor releases from version 3.0 to version 3.6, which spans a time frame of six years. The resulting feature models are relatively shallow with a maximum depth of 5 (when considering the root as 0). The number of features in the models ranges from 177 features in version 3.1 to 621 features in version 3.6 (Fig. 15). We then applied the model transformation from Section 5.1 to incrementally create the EvoFM models up to one that covers all versions from 3.0 to 3.6. The transformation also creates the resulting EvolutionPlan, which can be exported to MS Excel.

Fig. 15 shows our results. The upper part shows the number of features in each version and the sum of features over all versions. The lower part shows the resulting number of EvoFeatures and the number of uses of the EvoOperator moveTo, which indicates that features are moved within the model (due to our feature model interpretation the only other EvoOperator which can occur is the requires operator which is not considered here). As visible in the chart, the number of EvoFeatures and EvoOperators is significantly lower than the number of features but increases non-linear in the last version. After four versions (version 3.3) the EvoFM contains 36 EvoFeatures and 9 moveTo-operators which is easy to handle. After seven versions (version 3.6) EvoFM contains 109 EvoFeatures and 50 moveTo-operators. This is still a significant reduction of complexity compared to the feature model itself but a major increase compared to version 3.3. The main reason for this is that in earlier versions (3.0–3.3) large changes and refactoring occurred which can be subsumed into large clusters, while in later versions mainly punctual changes occur which cannot be clustered further. For instance, in version 3.6, 39 new features were added which resulted in 31 new clusters, i.e., most new clusters contain only a single feature. This corresponds with the general theory about evolution (Lehman and Belady, 1985) that large changes occur mainly on the early versions of a major release while the later minor versions contain only punctual changes. According to that, the ratio between EvoFeatures and features should increase again with the next major release of Eclipse.

Fig. 16 shows the resulting EvolutionPlan. The left hand side shows the complete EvolutionPlan for versions 3.0–3.6 with all EvoFeatures and moveTo-operators to give an idea of the configurations, the right hand size shows an extract in larger resolution. The selection of an EvoFragment or EvoOperator is indicated by green (darker) color. While the representation on the right has some similarities with visualizations like (Lanza, 2001), the difference is that each row represents a fragment. Ideal tool support can, for instance, show the content of an EvoFragment in a separate view if an EvoFragment is selected and allow to modify fragments. Also, since EvoFM is a specialized kind of feature model, existing interactive techniques (Bottrweck et al., 2009) can be applied to reduce its complexity.

As visible on the right hand side in Fig. 16, the sequence of EvoConfiguration decisions for the single elements (i.e., the combinations of selections/deselections within each row) occur in many different variations. Altogether 30 different combinations of decisions can be found (in the single rows), including that, e.g., some EvoFeatures are selected, then deselected, and later on selected again (e.g., EvoFragment P_1. org.eclipseplatform.source.linux.gtk. amd64).

As described in Section 5.1, the model transformation generates the names of EvoFeatures from the root features in the fragments. As in our example the feature names are names of Eclipse Features (marked with prefix F_) and Eclipse Plugins (marked with P_), the fragment names are rather technical. Of course, developers can change the generated names if desired.

In summary, the study shows that the detection of clusters works for real-world feature models covering evolution history of many years. In addition, it shows that indeed various combinations of decisions occur for the single fragments, which justifies the usage of an evolution plan. Although the implementation has not yet been optimized for speed or resource consumption our prototypical implementation required only 31 seconds for the calculation of the largest EvoFM (including output of reports). This indicates that the performance is sufficient for large FMs.
6.2. Threats to validity

When discussing the evaluation we have to consider multiple threats to validity, i.e., construct, internal, and external validity.

6.2.1. Construct validity

The idea of using generated (automatically constructed) feature models from the Eclipse platform is a threat to validity. The resulting feature model is likely to differ to some extent from a feature model which would be manually created and maintained for the same project. Therefore the kinds of operation on the models and the kinds of model elements cannot be directly compared to a real-world model of the same size.

We chose to use this approach since lack of empirical data in product line engineering is a well-known problem (Heider et al., 2010). Apart from that, the size of the feature models was more important for us than the detailed model semantics as we do not use it to derive general conclusions about feature model evolution from the model (like Lotufo et al., 2010) – we rather show that our modeling approach can basically deal successfully with very large feature models. Thus, this can be considered a minor threat to validity.

6.2.2. Internal validity

Some other factors that may affect the conclusions of our experiment are the types of operators. The example models used for the evaluation only use the moveTo operator. A usage of additional operator types might lead to different numbers. As shown in Fig. 2, many EvoOperators go along with moving features or feature groups (e.g., moving some mandatory features into a feature group). As we support complex operators, such cases would not lead to increased number of EvoOperators, only the type of EvoOperator would change (e.g., replacing a moveTo with a moveToFeature-Group). The only changes which would lead to increased number of EvoOperators are those which are neither related to moving features nor to adding or removing features, e.g., changing a single feature from mandatory to optional or adding/removing a single cross-tree constraint. It is not likely that these kinds of changes occur in a very high number (as described in Lotufo et al. (2010), more than 90% of the detected changes were related to adding, removing, moving or renaming features. Less than 10% were classified as a combination of changes).

An issue to be handled separately is the renaming of features. This is not handled automatically since without further domain knowledge about a feature it is impossible to decide if a feature has been removed (and a new one added) or if it was renamed. Hence, this has to be specified manually. However, it might be possible to find candidates for renaming by heuristics, e.g., by analyzing their attributes and relationships with other elements. Nevertheless, considering renaming of features would not increase the overall number of fragments and EvoOperators in EvoFM as renaming is specified by a single EvoOperator while removing a feature and adding a new one requires at least one fragment (for the new one) but often an additional split of existing fragment (for removing the old one) and possibly an additional EvoOperator (for moving children of the old feature to the new one).

6.2.3. External validity

General applicability of the results is yet to be investigated. The process of planning evolution of a product line may not proactive in all domains. In some cases, such pre-planning may not be adequate or may even be impossible to deal with new customer requirements.

Apart from that, EvoFM is rather focused on feature-tree representation of variability models. In many domains, such a tree-based model may be substituted by graph-like representations. The implications for EvoFM in these domains is yet unknown. The current set of operators, node types and constraints may not be adequate for all domains.

7. Summary and outlook

This paper presented EvoPL, a model-driven approach to specify and manage product line evolution on feature level (described in a special feature model, which we call EvoFM). We treat the evolution as a sequence of versions of a feature model (associated with different points in time). Abstraction is achieved by a concept to subsume multiple related changes in the feature model and associate them with a single rationale description. EvoPL is implemented by model transformations (addressing G4). A first transformation allows to automatically extract an EvoFM from a given sequence of feature models (i.e., previous evolution). It can then be refined and extended with, e.g., rationale descriptions. In addition, we discussed analysis of evolution based on EvoFM (addressing G5). Finally, a model transformation generates new (or updated) feature models based on the evolution specified on EvoFM level.

EvoPL supports abstraction (EvoFragments in Section 4.1), complex dependencies between fragments (EvoFM in Section 4.2), and order-independent planning (Evolution Plan in Section 5.2). Together with the support for rationale and a notion of time it addresses all requirements identified in Section 2.3.

Model transformations allow the automated generation of an Evolution Plan from existing versions (Section 5.1) as well as the generation of future version from an Evolution Plan (Section 5.2). As the model transformation is incremental, it provides maximal flexibility and supports to add additional existing feature models to the Evolution Plan at any time. The Evolution Plan provides a very compact and efficient visual representation of previous and planned evolution.

All described models and model transformation have been implemented with EMF and ETL. In addition, some first prototypical visual tool support is available. We evaluated our approach by applying it to a large real-world data from the Eclipse project (Section 6.1).

Future work will include the implementation of more integrated visual tool support and the future application to further real-world data and settings. We are currently addressing the propagation of changes down to the implementation level for product lines in the embedded system domain (see Section 5.4) and further investigating the specification of goals, rationale, and dependencies between them for evolution planning (see Section 4.4).

We also plan to extend the analysis of historic evolution based on the Evolution Plan. This includes the comparison of the data gathered from the Eclipse project with other data, e.g., the Linux Kernel feature model described in Lotufo et al. (2010). In the long term, we hope to identify common patterns in the evolution of product lines.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 10/CE/11855 to Lero – the Irish Software Engineering Research Centre, http://www.lero.ie/.

References
