Back to the Future: Avoiding Paradoxes in Feature-Model Evolution

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

Christoph Seidl
TU Braunschweig
Brunswick, Germany
c.seidl@tu-bs.de

Thomas Thüm
TU Braunschweig
Brunswick, Germany
t.thuem@tu-bs.de

ABSTRACT

A Software Product Line (SPL) captures families of software products and its functionality is captured as features in a feature model. Similar to other software systems, SPLs and their feature models are subject to evolution. Temporal Feature Models (TFMs) are an extension to feature models that allow for engineers to model past feature-model evolution and plan future evolution. When planning future evolution of feature models, multiple evolution steps may be planned upfront but changed requirements may lead to *retroactively* introducing evolution steps into the planned evolution or changing already planned steps. As a consequence, inconsistencies, which we denote as *evolution paradoxes*, may arise leading to invalidity of already modeled future evolution steps. In this paper, we present first steps towards allowing to introduce intermediate evolution steps into planned evolution while preserving consistency of all future evolution steps. To this end, we outline a method to define and check model evolution consistency rules. Using this method, engineers are allowed to introduce intermediate feature-model evolution steps whenever these changes preserve the evolution consistency rules.

ACM Reference Format:
Michael Nieke, Christoph Seidl, and Thomas Thüm. 2018. Back to the Future: Avoiding Paradoxes in Feature-Model Evolution. In *22nd International Systems and Software Product Line Conference - Volume B (SPLC '18), September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236405.3237201

1 INTRODUCTION

A *Software Product Line (SPL)* is an approach for large-scale reuse for products of software families [15, 16]. On an abstract level, functionality of these products is captured in terms of *features*. Features can be organized in variability models, such as feature models. A feature model is a tree-like notation expressing relations between features [5] and additional dependencies can be expressed using *cross-tree constraints* [8].

Similar to other software systems, SPLs are subject to evolution due to changed requirements [9, 13]. Optimally, feature-model evolution serves as starting point for evolving the entire SPL [12]. As

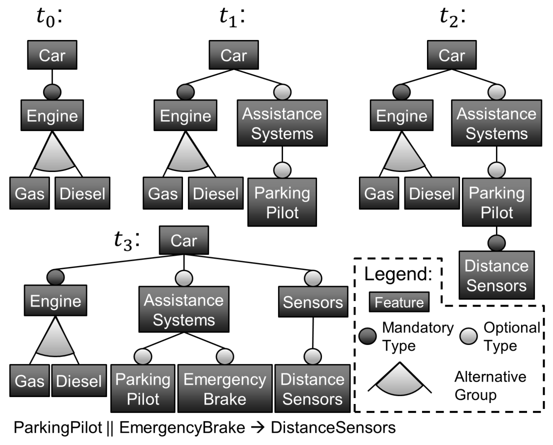

Figure 1: Planned future evolution steps of a car SPL

SPLs usually capture large families of software systems, SPL evolution is a large endeavor and may cost a significant amount of time and money. Thus, SPL evolution should be planned thoroughly as opposed to ad-hoc activities. When planning SPL evolution, engineers might model multiple future evolution steps. For instance, Figure 1 depicts the feature model of a car SPL (t_0) and its planned future evolution steps (t_1 – t_3). In the first planned evolution step at t_1, two features, `AssistanceSystems` and `ParkingPilot`, are integrated. For the evolution step at t_2, engineers plan to extract a feature `DistanceSensors` from the `ParkingPilot` feature to allow other future assistance systems to use these sensors. The evolution at t_3 introduces a new assistance system feature `EmergencyBrake`, which also makes also use of the `DistanceSensors`. Additionally, to improve the feature-model structure, a new feature `Sensors` is introduced, which should group all sensor features. Accordingly, the `DistanceSensors` feature is moved under the `Sensors` feature.

During executing this evolution plan, changed requirements force engineers to make an ad-hoc change to the evolution yielding an intermediate evolution step: The `ParkingPilot` feature is removed after t_1 as this car is supposed to be a low budget car and unnecessary features should not be integrated. However, at t_2 the `DistanceSensors` feature is introduced as child of `ParkingPilot`. Furthermore, at t_3 it is moved under the `Sensors` feature and also required by the `EmergencyBrake` feature. Thus, if the `ParkingPilot` feature is removed at t_1, the model becomes inconsistent starting at t_2 as the feature `DistanceSensors` does not have a parent anymore.

As illustrated by this example, changed requirements or if evolution had to be modeled differently than expected, engineers have to introduce intermediate evolution steps. However, other evolution steps that are planned for later points in time rely on the evolution that was originally planned. As a consequence, performing the intermediate evolution step may result in inconsistencies. We denote inconsistencies that affect already modeled future evolution steps and emerge by introducing an intermediate evolution step or changing an already planned evolution step as *evolution paradoxes*. When an evolution paradox has been introduced, engineers end up with an inconsistent model when going back to a future evolution step that is affected by that paradox. Up to now, we avoid these evolution paradoxes through limiting the functionality of editors in our tools.

In this paper, we present our ongoing work to overcome these limitations allowing engineers to introduce intermediate evolution steps or change existing evolution steps without introducing evolution paradoxes. In particular, we make the following contributions:

- We describe how to apply our previous work for future planning of feature-model evolution and we create awareness for the problem of evolution paradoxes.
- We outline a method to define evolution consistency rules and allowing to model intermediate evolution steps while avoiding evolution paradoxes.

2 BACKGROUND

In previous work, we devised the concept of Temporal Feature Models (TFMs) to capture feature-model evolution and provided a corresponding metamodel [11]. The basis of TFMs is the concept of a *temporal element* that is defined by a *temporal validity* $\vartheta = [\vartheta_{since}; \vartheta_{until})$ – a right-open interval of dates stating the timespan in which the element is valid. To allow feature-model evolution, we modeled each feature-model element that can possibly change as temporal element. This includes features, feature names, groups, feature types (*optional / mandatory*) and group types (*or / alternative*). To move features in the tree during evolution, we also modeled relations between groups and features as temporal elements and, thus, as own entities. With TFMs, we are able to model arbitrary feature-model evolution and capture its entire evolution history in one model.

To enable engineers to model TFMs, we implemented an editor hiding the complex details of the TFM metamodel within the tool suite DarwinSPL [10]. Using this editor, engineers are able to set a date in the editor and perform common changes to the feature model. These changes are captured as temporal elements in the background and, thus, registered as feature-model evolution.

Using TFMs to model feature-model evolution, the evolution of each model element itself is modeled in contrast to modeling disconnected individual feature models for each evolution step. As a consequence, we are able to extract more knowledge from the evolution history as we do not have to rely on model diffing approaches or interpolations. Additionally, more complex and more efficient analyses may be possible, e.g., to find feature-model anomalies [2].

3 SAFELY TRAVELING BACK TO THE FUTURE

When planning evolution of an SPL, many requirements and stakeholders must be considered. For instance, if a car manufacturer plans a new car version, management, politics, laws, economical, and ecological aspects need to be considered. As a consequence, evolution has to be planned a long time in advance possibly for multiple

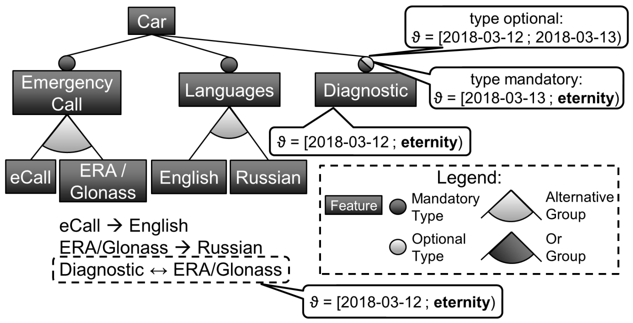

Figure 2: Planned future evolution of a car SPL as TFM

evolution steps, ideally starting with the feature model [12]. However, when engineers retroactively introduce intermediate evolution steps or change already planned evolution steps, they potentially introduce evolution paradoxes resulting in inconsistencies of future evolution steps. In the following, we describe how planning of future feature-model evolution can be performed using TFMs, what types of evolution paradoxes we consider and how to avoid them.

3.1 Modeling Future Feature-Model Evolution

TFMs make use of temporal elements to capture feature-model history (cf. Section 2). Using the same concept but dates in the future for temporal validities, we are able to plan future evolution. We denote the entirety of the feature-model evolution, i.e., the past history and the planned future evolution, as *evolution timeline*. We use a single TFM to plan the evolution of the running example in Figure 2. We assume t_0 to be the date before evolution and t_1 – t_3 to be the planned future evolution steps. The temporal validities of the relevant parts are illustrated as annotations in the diagram. For brevity, the parent-child edge between `ParkingPilot` and `DistanceSensors` is omitted. With the TFM editor of DarwinSPL, it is possible to model future evolution by setting the date of the editor to a future date. Until now, we introduced an artificial limitation only allowing to perform changes to the last evolution step to prevent the introduction of evolution paradoxes. However, given appropriate analyses, using TFMs allows us to search for potential evolution paradoxes as the relation between feature-model elements is still maintained when introducing intermediate evolution steps (unlike with unconnected feature model states as in Figure 1).

3.2 Evolution Paradoxes in Models

Planning SPL evolution is a complex task. During executing an evolution plan, new or changed requirements may demand for adhoc changes diverging from the original plan. As a consequence, already planned evolution steps have to be changed and intermediate evolution steps are introduced. For instance, in the running example, three future evolution steps are planned, the first evolution step planned for t_1 has been changed retroactively, reverting the introduction of the `ParkingPilot` feature. The evolution step planned for t_2 requires the introduction of `ParkingPilot` as the new `DistanceSensors` feature has been added as its child. As a consequence, an evolution paradox has been introduced as `DistanceSensors` does not have a parent anymore at t_2. However, when using individual feature models for each evolution step, modifications of intermediate evolution steps or changed pre-planned evolution steps are not transferred to future evolution steps. Thus, individual feature models are not suitable for planning future evolution and detecting evolution paradoxes would be very complex. With a TFM, modifications of an introduced intermediate evolution step are transferred to future evolution steps. For instance, the feature `ParkingPilot` at t_1 and t_2 is the same model entity. When removing this feature at t_1 in a TFM, it is also removed for all following evolution steps. Thus, we argue to use TFMs for modeling, planning, and reasoning about feature-model evolution.

For plain TFMs, the set of evolution operations that may principally lead to evolution paradoxes is limited. For instance, when deleting a feature, it may be checked whether this feature has children in future evolution steps. Thus, it would suffice to check these operations. However, when considering feature-model extensions, such as feature attributes and constraints for these attributes, checking for evolution paradoxes becomes significantly more complex. For our concepts to also be applicable to such extensions, we consider evolution paradoxes on the level of models and metamodels.

In model-driven software development, metamodels specify the abstract syntax of models, including classes and attributes, and their relations, i.e., references between classes and cardinalities of those relations [1]. Additional constraints may specify well-formedness of models, e.g., with the Object Constraint Language (OCL), to enforce modeling rules not captured directly in the metamodels structure. On metamodel level, evolution paradoxes occur if changes are performed to an evolution step that cause model constraints to be violated for already pre-planned future evolution steps. Due to the lack of a notion of time in existing constraint languages, they are not sufficient to describe the well-formedness in the presence of evolution or to avoid evolution paradoxes.

3.3 Temporal Invariants

Allowing intermediate evolution steps without introducing paradoxes requires to identify constraints that should not be violated during evolution. In general, metamodels and constraints are not aware of evolution or time. To overcome this limitation, we define the term of *temporal invariants*. A temporal invariant is a constraint that must hold for the entire evolution timeline of a model. Such constraints may be defined using a generic descriptive constraint language similar to OCL. In contrast to existing constraint languages, in such a language the temporal validity of temporal elements has to be accessible. When checking temporal invariants at one point in time, only elements that are temporally valid at that point in time are considered. We consider two more specific subtypes of temporal invariants: *Reference validity constraints* and *temporal cardinalities*.

Reference validity constraints reason about the relation of temporal validities of the source and target temporal element of a reference. This is necessary in cases in which an element may only be temporally valid if another element is temporally valid as well. We consider three types of reference validity constraints:

- The source entity and the target entity may not exist without each other. We consider the temporal validities of the source entity and target entity to be equal, i.e., $\vartheta_{source} = \vartheta_{target}$.
- The target entity cannot exist without the source entity, e.g., when the source entity controls the lifecycle of the target entity such as with containment references of EMF Ecore. The temporal validity of the source entity may exceed the temporal validity of the target entity, i.e., $\vartheta_{source} \supseteq \vartheta_{target}$.
- The source entity cannot exist without the target entity. This is opposite to the second case and appears when the target-entity lifecycle is independent of the source entity, i.e., the reference must not be a containment reference of EMF Ecore. The source-entity temporal validity may be exceeded by the target-entity temporal validity, i.e., $\vartheta_{source} \subseteq \vartheta_{target}$.

Temporal cardinalities are similar to common metamodel cardinalities but checking conformance is more challenging as each point in time has to be considered. To be able to automatically derive reference validity constraints and temporal cardinalities, we plan to introduce a process transforming an ordinary metamodel of a particular notation to a temporal metamodel. The information gathered during the transformation could then be use to (semi-) automatically generate appropriate temporal invariants.

3.4 Verifying Evolution Consistency Rules

We plan to implement a generic method that provides the notion of temporal elements allowing to define temporal metamodels. Additionally, it provides the possibility to define temporal invariants. Using a set of temporal invariants, a concrete model instance, and using our method, we are able to verify whether these invariants hold.

For the verification mechanism, we plan to evaluate two approaches: The first approach iterates over all evolution steps of a model and verifies the constraints for each of these points in time. The second approach encodes TFMs as a Satisfiability Modulo Theories (SMT) problem, which could be beneficial for performance [3]. To this end, a distinct evolution variable needs to be introduced and constraints using this variable have to be defined. Then, we can use quantifiers to reason on the entire future evolution (timeline).

When performing intermediate evolution steps, our method can be used to verify whether temporal invariants still hold after the changes. For instance, a temporal invariant for TFMs is that each feature (except for the root) has a parent at each point in time the feature itself is temporally valid. To verify such invariants, we investigate two possibilities: First, we can perform the evolution operation at t_i on a copy of the model. Then, we check whether, for all evolution steps $> t_i$, the temporal invariants still hold. If this is the case, the original model is replaced by the copy and, if not, the evolution operation is not executed. For instance, when deleting the feature `DistanceSensors` at t_1 in the running example of Figures 1 and 2, we would copy the TFM and iterate over all points in time $\geq t_1$, verifying the invariant. Second, we analyze whether a certain evolution operation may principally violate the temporal invariants. If this is the case, we determine the conditions for the violation. For performed evolution operations, we then verify whether the conditions apply or not to prohibit or permit the operation execution. For instance, for the above mentioned invariant, we would determine that the delete feature operation may violate the invariant if the respective feature has a child feature in future points in time. In future research, we want to verify whether the identification of problematic evolution operations and the condition determination may be derived automatically. With our method, we would allow safe intermediate evolution without introducing evolution paradoxes.

4 RELATED WORK

Engels et al. studied how to preserve model consistency in presence of evolution [4]. They argue that it is unnecessary to verify the entire model but only changes affected by the evolution may have to be verified. However, they only specify concrete rules for their own statechart models and do not consider intermediate evolution.

Kehrer et al. studied how to automatically derive consistency preserving model evolution operations but they do not consider evolution paradoxes [7]. With model differencing approaches, changes performed during evolution to a model can be derived. Kehrer et al. present a method to derive edit scripts from comparing model differences [6]. When introducing intermediate evolution steps, such scripts could be derived and used to modify pre-planned future feature-model versions. These modified feature-model versions could then be checked for consistency. However, with TFMs, such scripts are not necessary as evolution operations are changes to temporal validities. These changes are automatically applied for future evolution steps. Moreover, differencing methods may be inaccurate in certain scenarios which is not possible with TFMs.

Schwägerl et al. present a method to combine version and variant management for SPLs [17]. To this end, they consider modifications to be performed on domain artifacts that are associated with a feature configuration. These changes can then be committed to a repository and respective changes are then related to this feature combination. Schwägerl et al. also assume to have a *multi-version domain model* for each of the used notations in the repository [17]. The concept of temporal metamodels in combination with a versioning system could be used for creating multi-version domain models. However, to the best of our knowledge, they do not consider what happens if multiple users define conflicting change operations which would be a similar problem to the evolution paradoxes.

Compared to using standard versioning systems, such as SVN or Git, TFMs are more flexible. Without branches, model evolution is performed in a linear fashion, i.e., without intermediate evolution steps. With branches, intermediate evolution steps are defined in new branches. To apply these changes to all future evolution steps, the newly created branch has to be merged into all future branches, possibly leading to errors or inconsistencies during the merge.

5 CONCLUSION

In this paper, we presented a concept for planning future feature-model evolution and allowing to introduce intermediate evolution steps without causing evolution paradoxes that violate consistency of pre-planned future evolution steps. With the concept of temporal invariants, we introduced the possibility to define constraints on the evolution timeline of a model. We defined temporal invariants in a general way to be applicable for arbitrary metamodels. We outlined a generic method to verify temporal invariants and to check whether performed evolution operations can be executed without introducing evolution paradoxes.

In our future work, we will investigate how to efficiently verify that intermediate model-evolution operations may be performed while preserving all temporal invariants for all model versions. We will implement our method to generate temporal metamodels for arbitrary base metamodels and to define and verify temporal invariants. This also allows us to investigate the consistent evolution of models linked to features. Moreover, we will investigate the possibility to define more complex and detailed temporal constraints using Linear Temporal Logic (LTL) [14].

ACKNOWLEDGMENTS

This work was partially supported by the Federal Ministry of Education and Research of Germany within CrESt (funding 01IS16043S), by the DFG (German Research Foundation) under SPP1593: Design For Future — Managed Software Evolution.

REFERENCES

[1] C. Atkinson and T. Kuhne. 2003. Model-driven development: a metamodeling foundation. *IEEE Software* 20, 5 (Sept 2003), 36–41.
[2] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.
[3] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77.
[4] Gregor Engels, Reiko Heckel, Jochen M. Küster, and Luuk Groenewegen. 2002. Consistency-Preserving Model Evolution through Transformations. In *UML 2002 — The Unified Modeling Language*, Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 212–227.
[5] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
[6] T. Kehrer, U. Kelter, and G. Taentzer. 2013. Consistency-preserving edit scripts in model versioning. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 191–201.
[7] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. 2016. Automatically Deriving the Specification of Model Editing Operations from Meta-Models. In *Theory and Practice of Model Transformations*, Pieter Van Gorp and Gregor Engels (Eds.). Springer International Publishing, Cham, 173–188.
[8] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-world Feature Models and Product-line Research?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 291–302.
[9] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 136–150.
[10] Michael Nieke, Gil Engel, and Christoph Seidl. 2017. DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-aware Software Product Lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '17)*. ACM, New York, NY, USA, 92–99.
[11] Michael Nieke, Christoph Seidl, and Sven Schuster. 2016. Guaranteeing Configuration Validity in Evolving Software Product Lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '16)*. ACM, New York, NY, USA, 73–80.
[12] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. 2013. Feature-oriented Software Evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*. ACM, New York, NY, USA, Article 17, 8 pages.
[13] Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wąsowski. 2012. Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development (FOSD '12)*. ACM, New York, NY, USA, 62–69.
[14] A. Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57.
[15] K. Pohl, G. Böckle, and F.J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer Berlin Heidelberg.
[16] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (01 Oct 2012), 477–495.
[17] Felix Schwägerl and Bernhard Westfechtel. 2017. Maintaining Workspace Consistency in Filtered Editing of Dynamically Evolving Model-driven Software Product Lines. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*. SciTePress, 15–28.