Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution

Daniel Hinterreiter
Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz
Austria
daniel.hinterreiter@jku.at

Michael Nieke
Technische Universität Braunschweig
Germany
nieke@isf.cs.tu-bs.de

Lukas Linsbauer
Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz
Austria
lukas.linsbauer@jku.at

Christoph Seidl
Technische Universität Braunschweig
Germany
c.seidl@tu-braunschweig.de

Herbert Prähofer
Institute for System Software
Johannes Kepler University Linz
Austria
herbert.praehofer@jku.at

Paul Grünbacher
Inst. Software Systems Engineering
Johannes Kepler University Linz
Austria
paul.gruenbacher@jku.at

Abstract

A feature model (FM) describes commonalities and variability within a software product line (SPL) and represents the configuration options at one point in time. A temporal feature model (TFM) additionally represents FM evolution, e.g., the change history or the planning of future releases. The increasing number of different TFM notations hampers research collaborations due to a lack of interoperability regarding notations, editors, and analyses. We present a common API for TFMs, which provides the core of a TFM ecosystem, to harmonize notations. We identified the requirements for the API based on systematically classifying and comparing the capabilities of existing TFM approaches. Our approach allows to work seamlessly with different TFM notations to perform, track, analyze and replay evolution. Our evaluation investigates two research questions on the expressiveness (RQ1) and utility (RQ2) of our approach by presenting implementations for several existing FM and TFM notations and replaying evolution histories from two case study systems.

CCS Concepts • **Software and its engineering → Software configuration management and version control systems**.

Keywords Software product lines, evolution

GPCE '19, October 21–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6980-0/19/10…$15.00
https://doi.org/10.1145/3357765.3359515

ACM Reference Format:
Daniel Hinterreiter, Michael Nieke, Lukas Linsbauer, Christoph Seidl, Herbert Prähofer, and Paul Grünbacher. 2019. Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357765.3359515

1 Introduction

Feature modeling is widely used to describe commonalities and variability within a software product line (SPL) [6, 30]. A *common* feature model (FM) defines configuration options of an SPL at one point in time. However, as an SPL needs to evolve, temporal perspectives on FMs become important. For instance, companies nowadays have to maintain product lines in different stages (e.g., development, nightly builds, long-term support) meaning that they need to maintain revisions of individual features. In such a context it becomes crucial to understand the evolution history, to capture ongoing changes, or to plan and coordinate future development [3, 26, 27].

A *temporal feature model (TFM)* thus aims to capture the entire evolution timeline of a FM's configuration options and structure. Unlike other approaches that are concerned with the evolution of FMs and SPLs [8, 12, 14, 15, 20, 22, 28, 29, 33, 36], TFMs capture the information pertaining to evolution tightly integrated with the notation for feature modeling, which eases tracking and analyses of evolution. Several TFM approaches have been developed: DarwinSPL [24], DeltaEcore [37], EvoFM [2], Feature-Driven Versioning (FDV) [21], FORCE [15], and SuperMod [34, 35] extend FMs to represent different aspects and levels of evolution. They have significant commonalities but also offer specific capabilities for their primary use cases. For instance, while some approaches

focus on feature-based development and composition, others emphasize feature-driven planning of SPL evolution.

We see three main stereotypes for prospective users of TFM approaches: The *SPL Manager* models variability in a FM, e.g., when adding, modifying and removing features, and copes with evolution by tracking changes and recreating previous states of the model. The *SPL Tool Developer* devises and implements tools and FM analysis techniques, some of which cope with evolution. The *Notation Developer* wants to provide an FM or TFM notation to these other roles.

In a recently published systematic literature review on SPL evolution [19], the authors argue that "the SPL community needs to work together to improve the state of the art, creating methods and tools that support SPL evolution in a more comparable manner". However, such research collaborations are challenging, as the heterogeneity of the different TFM implementations impedes reuse and their specific capabilities are not well understood. In particular, while each of the TFM notations offers specializations, there are also individual shortcomings such as notational deficits or incompatibilities. As a result SPL managers have to make decisions on acceptable trade-offs and may be locked in with their choice of notation. SPL tool developers have to support a myriad of different notations while struggling with technical incompatibilities of tool interfaces. Finally, notation developers need to develop boilerplate infrastructure (e.g., editors and configurators) to make their notation usable.

To address these problems, we have devised a TFM ecosystem, with an API for (T)FMs at its core. We identified key requirements for the common API based on a systematic analysis and comparison of existing TFM approaches. We further provide new API capabilities beyond the functionality of existing FM APIs. Specifically, this paper contributes: (i) a survey and systematic comparison of six TFM approaches; (ii) the definition of requirements for a common API to harmonize the capabilities of TFM approaches; (iii) the design of a two-layer, multi-view architecture providing support for performing, tracking, analyzing and replaying FM evolution; and (iv) an evaluation of our common API based on implementations for existing (T)FM notations and a replay of existing evolution histories of two product lines.

Our paper is organized as follows: Sect. 2 explains our research method. Sect. 3 compares key characteristics of selected TFM approaches. Sect. 4 discusses the requirements for our TFM ecosystem. Sect. 5 explains its architecture. Sect. 6 evaluates the expressiveness (RQ1) and utilily (RQ2) of our approach. Sect. 7 concludes the paper and provides an outlook on future work.

2 Research Method

While studies exist on various notations for FMs and their capabilities [1, 6, 16, 30, 32, 40] no such comparison exists for TFMs to the best of our knowledge.

Hence, we systematically analyzed and compared the existing TFM approaches to identify the key functional and qualitative requirements for our common TFM API. Specifically, our research was carried out as follows:

Literature Study. Based on our experience in SPLs, software evolution, and variation control systems, we started our search with existing survey papers [17, 19] and then used snowballing [4] to identify further yet undiscovered papers relevant for our context of TFMs. However, we did not further include remotely related approaches addressing specific aspects of co-evolution in SPLs (e.g., [11, 18]).

Selection of Subject Approaches. We included TFM approaches as subject approaches if their emphasis is on problem space evolution of variable systems and if they aim at capturing the evolution timeline of FMs' configuration options. Overall, we included six approaches in our comparison [2, 15, 21, 24, 35, 37], covering the essential approaches in this area of research. In some cases (e.g., [15]) several generations of subject systems exist, for instance, when a more advanced system was developed based on an earlier prototype. We chose the most mature generation for our comparison.

Definitions of Characteristics. We identified important properties for comparing the TFM approaches. In particular, we started with a core set of characteristics based on our experience as researchers in the subject matter. We iteratively refined our set of characteristics when reviewing and structuring the subject systems' capabilities, inspired by existing approaches on developing taxonomies (e.g., [23]). This also included harmonizing the terminology in some cases.

Classification of the Subject Approaches. All authors individually assessed the subject systems using the characteristics as a guideline. The individual classifications were then consolidated in a common classification. The authors carefully discussed and resolved all cases of disagreement.

Requirements Elicitation. We compared the different subject approaches to identify common characteristics. Several subject systems also describe industrial usage scenarios that we used to reflect on these characteristics and to identify requirements for the common TFM API.

API Design and Implementation. Based on these requirements we devised the two-layer, multi-view architecture of our common API, which supports the different views for the stereotypes defined above.

Evaluation. Our evaluation was guided by two research questions. We investigated the expressiveness (RQ1) of our TFM ecosystem by implementing it for five existing (T)MF approaches. This included developing API interfaces for three conventional feature modeling tools to evaluate support for exchanging models and transformations between models of conventional and TFM-based tools. We further assessed the utility (RQ2) of our approach by replaying evolution histories of two systems.

3 Existing TFM Approaches

We briefly describe the selected TFM subject approaches and compare them using important characteristics of TFMs.

3.1 Approaches

DeltaEcore / Hyper-Feature Models. DeltaEcore is a tool suite for the integrated management of variability in space and time (i.e., configuration and evolution) in SPLs [37]. The approach uses *Hyper-Feature Models (HFMs)* [39] and *evolution delta modules* [38] to capture evolution in the problem and solution space, respectively. HFMs permit the definition of features with arbitrary many *feature versions*, each representing that a feature's implementation has changed in a significant way. The respective changes to implementation artifacts are performed by evolution delta modules associated with feature versions, which perform transformations such as adding, removing, or modifying elements. Feature versions are assumed as incremental, i.e., v1.2 builds upon the state of v1.1. While feature versions can be arranged along branching development lines, merging is not supported. The approach also does not allow capturing evolutionary changes affecting the structure of FMs or feature characteristics.

DarwinSPL / Temporal Feature Models. DarwinSPL is a tool suite for modeling evolving context-sensitive SPLs [24]. DarwinSPL's TFMs [26] extend DeltaEcore's HFMs by treating feature-model evolution as first-class entity. TFMs facilitate evolution of the entire feature-model structure, such as feature existence, locations of features and groups, or variation types. This is achieved by assigning a *temporal validity* to each element of the FM. A temporal validity is a right-open interval $\vartheta = [\vartheta_{since}, \vartheta_{until})$ of two points in time where elements are valid within this interval. DarwinSPL provides a TFM editor that hides model complexity from its users by employing an *evolution slider*, which allows to select and display snapshots of the FM at different points in time. Changes to the FM are saved automatically in the background using the time of the evolution slider. Besides tracking the evolution history, TFMs allow for planning future FM evolution and retroactively introducing intermediate evolution steps. Due to the complexity of intermediate evolution steps, the editor currently only supports changes to the most recent step. Moreover, each element has exactly one temporal validity and may not be valid in several lifespans. TFMs do not support branching of evolution timelines and thus merging is also not supported.

EvoFM. EvoFM [2] is primarily intended to support long-term feature-oriented planning of product portfolios and analysis of evolution plans but can also be used for reviewing past evolution. Specifically, the approach emphasizes the commonalities and variability between the models over time and uses evolution FMs (EvoFMs) for modeling evolution. Following this idea, a FM of the product line at a specific point in time corresponds to a specific configuration of the FM. Mappings are maintained between EvoFM and the actual FM of the product line to enable the derivation of a FM by configuring an EvoFM. Visualizing these different configurations over time then allows to create an evolution plan giving an overview on the evolution of the product line. Finally, the approach also presents a catalogue of change operators for FMs to address structural changes in the FM that go beyond the presence or absence of features. In particular, the catalogue covers changes at the level of features (add, remove, replace, move, rename, make optional, make mandatory, ...) and changes on feature groups (add group, convert to alternative, replace constraint, ...).

Feature-Driven Versioning. The Feature-Driven Versioning (FDV) [21] approach aims to enable traceability during SPL development. For this purpose, FMs are extended by two types of versions for each individual feature: the *feature container version* represents the revision of the implementation artifacts associated with the feature. It is incremented whenever these artifacts change. The *feature logical version* represents the revision of a feature's characteristics (e.g., name and variation type) and the characteristics of its subtree. It increments in case of changes to the feature (or its subtree) or when its feature container version increases. Both types of versions are represented numerically and are (implicitly) assumed as sequential, meaning that neither branching nor merging is supported. With the focus on the variability model in the problem space, the evolution of implementation artifacts in the solution space is mentioned but not explicitly supported. Hence, FDV is able to represent *that* a feature or its implementation have changed but not *how*.

FORCE. The FORCE feature modeling language and tool suite have been developed for supporting feature-oriented software development in large industrial SPLs [31]. The FORCE modeling language supports group features, feature cardinalities, and various types of feature constraints. As an extension for reducing the complexity in large-scale systems FORCE introduced components, i.e., configurable units as proposed in the Common Variability Language [9]. Each unit contains a FM representing the variable characteristics of a particular component. Furthermore, FORCE supports modeling in multiple spaces and mappings of features between different modeling spaces. Features in FORCE are mapped to the source code implementing them via the variation control system ECCO [17]. Regarding evolution, the FORCE modeling language supports feature versions and FM versions [15]. It uses operations known from distributed version control systems and offers capabilities for checking out, committing, pushing, and pulling *features*. Specifically, revisions of features and FMs allow tracking the evolution of a FM and mapped artifacts over time. Each change to the FM is tracked by creating a new revision of the FM as well

Table 1. Comparison of Subject Approaches.

Characteristic	DeltaEcore	DarwinSPL	EvoFM	FDV	FORCE	SuperMod
Unit of Evolution	Version (FI)	Time (FM, F) Version (FI)	Feature[D]	Version (FM, F, FI)	Revision (FI) Version (FM, F)	Revision (FM, F, FI)
FM Evolution	–	+	o[E]	o[F]	+	+[I]
Implementation Artifact Evolution	+	+	–	o[F]	+	+[I]
Mapping Evolution	o[A]	+	–	–	+	+[I]
Configuration Evolution	o[B]	+	–	–	o[G]	–
Branching	+	o[C]	–	–	+	+[J]
Merging	–	–	–	–	o[H]	+

[A] Adds mapping for new evolution delta module, which references old evolution delta modules.
[B] Evolution in configuration by selecting features and one concrete version per feature.
[C] Branching supported for versions of FI, but not for FM's evolution timeline.
[D] Uses FM notation for planned evolution, e.g., optional feature added or deleted at some point.
[E] Explicitly modelled as future evolution plan.
[F] Can represent that a feature changed but not how it was changed.
[G] Configuration in context with FM version.
[H] Work in progress.
[I] Versioned with entire SPL, but can only retrieve one state at a time.
[J] Uses push/pull instead of branching.

as new revisions for the changed features. Moreover, feature revisions together with feature-to-code mappings allow tracking changes to implementation artifacts in the artifact repository. The FORCE modeling tool allows reviewing the evolution history. Users can use a *version slider* to inspect changes to the FM and the mapped artifacts or to retrieve earlier snapshots of the FM.

SuperMod. SuperMod [34, 35] is a tool for collaborative, model-driven software product line engineering. It combines concepts from distributed version control systems (such as Git), SPL engineering, and model-driven engineering. SuperMod provides operations familiar from version control systems, such as checkout or commit, and keeps revisions of the model-driven SPL under development. Every revision represents an evolutionary state of the model-driven SPL and stores its FM and its implementation artifacts. The FM is versioned by the revision graph, i.e., the FM can be retrieved by selecting a revision. The implementation is versioned by both the revision graph and the FM, i.e., the implementation can be retrieved by first selecting a revision and then a configuration. This is achieved by assigning visibilities to elements of both the FM and the implementation. Elements of the FM have visibilities attached that refer to revisions of the revision graph. Elements of the implementation have visibilities referring to both revisions of the revision graph and features of the FM.

3.2 Comparison

Table 1 summarizes the subject approaches regarding the following evolution characteristics:

Regarding the *unit of evolution*, i.e., the elements of which changes are tracked, the approaches work at different levels of granularity including feature models (FM), individual features (F), and feature implementations (FI). The approaches also support different kind of temporal units (e.g., time vs. version numbers).

With respect to *FM evolution*, i.e, structural changes to problem space FMs, DarwinSPL, FORCE and SuperMod provide full support for important change operations on FMs [2, 5, 8, 10], while other approaches suffer from some limitations summarized in the legend.

Full support for *implementation artifact evolution* – can the approaches represent changes to artifacts realizing the features? – is limited to DeltaEcore, DarwinSPL, and FORCE. *Mapping evolution* – can the mappings between the problem space features and their solution space artifacts evolve over time? – is only provided by DarwinSPL and FORCE. Only DarwinSPL provides full support for *configuration evolution*, i.e., it allows different versions of configurations.

DeltaEcore, FORCE, and SuperMod offer support for *branches* or clones from the main development line. SuperMod is currently the only approach considering *merging* changes from branches or clones to the main line.

4 Requirements

The definition of requirements was guided by two goals: harmonizing conceptual differences and improving (T)FM tool interoperability.

Harmonizing Conceptual Differences. Harmonizing the existing approaches for TFMs is concerned with making different notations uniformly accessible. To achieve this, the TFM API needs to support relevant common capabilities extracted from the inspected subject systems. Specifically, we investigated the individual characteristics and operations, and extracted those identified as shared commonalities or best practices. At the same time some characteristics are subject to limitations in some of the investigated systems. For example, some approaches restrict the number of variability

groups per feature, whereas others allow an arbitrary number. We thus tried to strike a balance between harmonizing concepts of all approaches and rectifying shortcomings of individual approaches.

Improving (T)FM Tool Interoperability. Most of the subject approaches provide tool prototypes supporting the proposed TFM notations. Additionally, a wide range of tools exists for non-temporal FMs [30]. However, support for exchanging models and transformations between models of existing tools and TFM-based tools is largely unexplored. For instance, an important use case is to transform a TFM to a non-temporal FM. This would allow, e.g., to utilize established analysis methods for non-temporal FMs on non-temporal instances of a TFM. Therefore, our common TFM API requires capabilities for interoperability, which map TFM API operations to specific operations of existing approaches (cf. Table 2).

SPL modelers using existing feature modeling approaches benefit from a common TFM API, as it allows extending existing tools with temporal feature modeling support, even without extending their own notations. Note that, unlike with an (incomplete) exchange format, our API does not lose information even if only part of the functionality is realized: The implementers of the API have full control over how information in the underlying native format is managed, e.g., to create sensible defaults or maintain existing notation-specific values, even if the peculiarities of a notation are not realized in the API.

Our comparison of the different approaches identified common characteristics but also differences, which served as a starting point for deriving eight specific requirements based on our two goals. We also considered industrial usage scenarios reported for several of the subject systems when defining the requirements. As explained above, our analysis focusses on TFM approaches that emphasize the problem space perspective of evolution of variable systems. This means that our requirements currently do not cover the evolution of implementation-to-artifact mappings, a capability provided by several of the subject approaches. We plan to address this in future work to also take into account the experiences with our initial solution.

R1: Creating and Initializing FMs. The TFM ecosystem needs to allow creating common (non-temporal) FMs, possibly by already adding an initial set of features and dependencies (mandatory features, optional features, alternatives, etc.). Furthermore, it is important to support adding, removing, and changing constraints during this initialization phase.

R2: Creating and Evolving TFMs. Users need support for creating TFMs with support for temporal properties. This is achieved by operations allowing to define at what point in time a change becomes effective. For instance, a feature may be added with a certain time stamp or version.

R3: Evolving FMs with TFM support. Users need support to create FMs and apply evolution operations on them. A TFM shall be created automatically in the background to provide TFM support, i.e., the operations need to additionally create the respective TFM elements such as time stamps or versions of a change. Furthermore, the TFM ecosystem should allow to temporally offset specific evolution operations by shifting the temporal parameter forward or backward. This is of interest when using TFMs for future evolution, i.e., postponing planned evolution operations.

R4: Views for FMs and TFMs. The TFM ecosystem needs to provide means for requesting models in either FM view or TFM view on demand (using a common tree-structured representation). This allows a user, e.g., to depict temporal aspects of a TFM, such as a FM at a particular point in time, or to visualize changes between two revisions of FMs.

R5: Providing Access to the Evolution Operation History. The TFM ecosystem needs to record changes and provide this information to external systems or components. This can be achieved via notifications providing access to the performed evolution operations, which contain the information required for recording and reconstructing the change history of FMs. For instance, this would allow a third-party component to request information on all features added since a certain point in time.

R6: Replaying Evolution. The TFM ecosystem records changes, i.e., the type and order of evolution operations on a FM. Replaying means to re-apply a set of recorded operations to update a different version of a FM. This is needed, for example, when creating a patch based on one version of a FM and then applying those same changes on another version. This can be beneficial, for instance, if two developers concurrently modify the same FM in different branches and both variants then need to be synchronized.

R7: Analyzing Evolution. The evolution history provides a rich source of information of what kind of changes appeared, which parts of the FM were affected as well as when and in which order the changes were applied. Analyzing this information helps, for instance, to identify hotspots in the FM, i.e., features or parts of a FM with frequent changes. The ecosystem thus needs support for analyzing the evolution of TFMs. Realizing this requirement can benefit from existing methods [13, 24–26].

R8: Controlling Access to (T)FM Operations. The permission to change (T)FMs often needs to be limited to certain users or user groups. Access control is thus an important requirement for feature modeling, particularly in software ecosystems. The TFM ecosystem shall provide a central access point for handling and managing change requests. For instance, different rights are usually required for inspecting or manipulating a FM.

5 Temporal Feature Model Ecosystem

Based on these requirements we have devised a TFM ecosystem to harmonize different FM and TFM notations, which enables significant reuse of tools and analyses. To facilitate

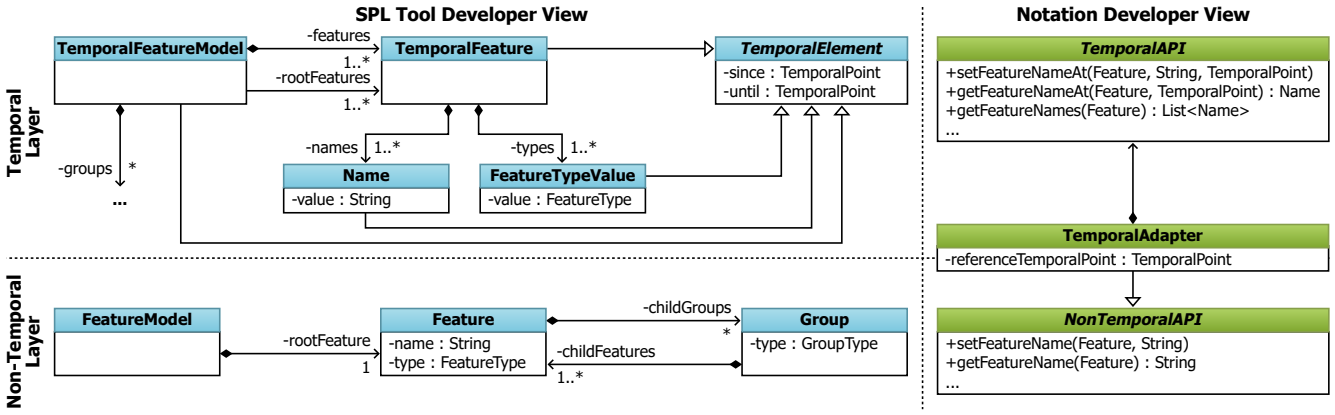

Figure 1. Excerpts from the core data abstractions of the TFM ecosystem.

the usage of temporal aspects for different stakeholders, we provide a two-layer architecture that allows for either implicit (transparent) or explicit access to information regarding evolution. Within each layer, we use different views for the stereotypes defined in Section 1, which enable to seamlessly perform, analyze, log and replay evolution operations. A *temporal point* can either be a date/time (e.g., "2019-02-26, 13:00:00") or a revision (e.g., "BetaRelease"). Furthermore, we distinguish two types of changes to FMs: *modifications* do not consider evolution (e.g., setting a feature's name in a standard FM editor), whereas *evolution operations* [2, 5] are changes at a specific point in time (e.g., a rename-feature operation performed at a specific time).

Alongside the core TFM API, we provide a base implementation (e.g., core logic for extensions, editors), which together form the TFM ecosystem. In the following, we describe the conceptual part of the TFM API and refer interested readers to our GitHub repository (https://bit.ly/2OEMGjD) for its reference implementation. Figure 1 and Figure 2 provide an overview of the main data abstractions and their usage.

5.1 Layers

We define two layers, the *non-temporal layer* and the *temporal layer*, as the primary dimension of decomposition (Figure 1). Additionally, we define a *temporal adapter* connecting these layers to allow transparent usage of evolution functionality, i.e., without explicitly using the temporal layer.

Non-Temporal Layer. The non-temporal layer provides abstractions for the data and usage of FMs that are agnostic of evolution concerns, e.g., by allowing to change a feature's name independent of the concrete underlying notation. Its primary purpose is to abstract from the specifics of various notations of (common) FMs so that they can be used uniformly, especially when performing modifications. The non-temporal layer serves as basis for the TFM ecosystem, but may also be used directly to create, manipulate or analyze FMs, e.g., within editors. Modifications performed on the non-temporal layer have instantaneous effects and are not tracked as evolution, e.g., `Feature.setName(String)` directly sets the name instead of performing a rename-feature operation.

Temporal Layer. The temporal layer provides abstractions for data and usage of TFMs that make evolution concerns explicitly accessible, e.g., that a feature's type changed at a particular point in time. Its primary purpose is to enable querying and manipulation of information regarding evolution independent of the concrete TFM notation. Correspondingly, and in contrast to the non-temporal layer, queries to the temporal layer return collections of elements, e.g., `getAllFeatureNames(TemporalFeature)` returns all names with the respective temporal intervals in which they are valid. Additionally, it is possible to query elements valid at a specific point in time, e.g., `getFeatureVariationTypeAt (Feature, TemporalPoint)` returns the variation type valid at the given temporal point. Similarly, the manipulation of temporal elements is performed by giving a certain temporal point from which on the changes should be effective, e.g., `setFeatureNameAt(Feature, String, TemporalPoint)` changes the name of the respective feature from the given temporal point on.

When using a common FM in an SPL and performing FM evolution, the temporal layer utilizes the non-temporal layer, as the effects of an evolution operation are manifested through a modification on the non-temporal layer, e.g., renaming a feature on the temporal layer is manifested through calling the respective method to set a feature's name on the non-temporal layer, while still recording the time of change.

Temporal Adapter. The non-temporal layer is agnostic of evolution whereas the temporal layer provides explicit access on evolution concerns at the cost of exposing its users to the intricacies of evolution, e.g., that a feature may have more than one name over the course of its existence. While both layers address valid concerns, there is also the need for a solution allowing to manage evolution implicitly without exposing users to the intricacies of evolution, e.g., an SPL Manager performing a series of changes on an FM wants to

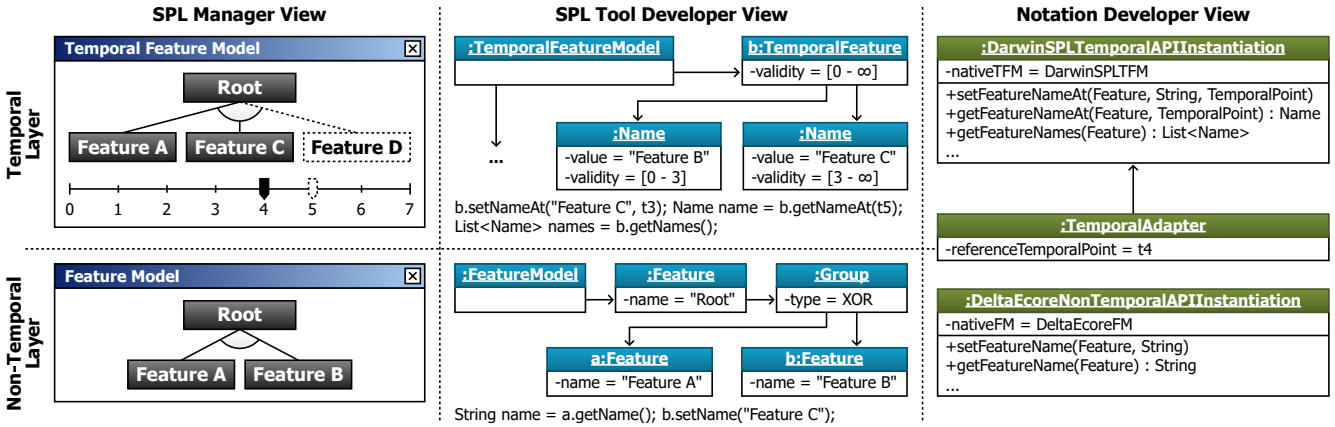

Figure 2. Example of the core abstractions when using the TFM ecosystem to manage SPL evolution with (T)FMs.

track the evolution history without being confronted with the resulting complexities in a TFM.

For this particular purpose, we provide a *temporal adapter* that offers the functionality of the temporal layer but mimics the interface of the non-temporal layer. Specifically, the temporal adapter stores a reference temporal point that is used for the next sequence of operations. Calls to the temporal adapter via non-temporal operations are then delegated to the respective operations on the temporal layer with the reference temporal point as an additional parameter to perform the adequate evolution operation. For instance, a call issued to `setFeatureName(Feature, String)` on the temporal adapter (i.e., non-temporal layer) is delegated to `setFeatureNameAt(Feature, String, TemporalPoint)` on the temporal layer using the reference temporal point as the last parameter.

As a result, when using a reference temporal point, a TFM can be treated equivalently to a common FM. The choice of how to set the reference temporal point is left to users of the temporal adapter. It it possible to set it explicitly, e.g., via a temporal slider as in our editor, or implicitly, e.g., by using the current date. This transparent use of TFMs allows for seamless integration into the infrastructure of common FMs, e.g., editors or analyses, independent of evolution. This means that engineers can benefit from capturing the evolution timeline without changing their existing tools. The information on evolution can later on be used by analyses or other tools using the temporal layer.

5.2 Views

As the secondary dimension of decomposition within the TFM ecosystem, we provide individual views suitable for the intentions of the three stereotypes (cf. Section 1) on each of the non-temporal and temporal layer.

The **SPL Manager View** utilizes a visual representation of the (T)FM (see Figure 2).

We provide full-fledged implementations of an editor and a configurator for the representations on both the non-temporal and the temporal layer. For the latter, users may select a particular temporal point via a slider for which the respective state of the FM is presented. When editing a FM on the temporal layer, the editor transparently instantiates and applies respective evolution operations, which create the appropriate structures in the TFM by using the temporal adapter. Hence, the intricacies of performing and tracking evolution are hidden entirely from the SPL Manager, who can still benefit from the underlying TFM, e.g., through tracking or analyzing evolution operations. As an SPL Manager does not manipulate a (T)FM programmatically but through an editor, we do not provide a separate view on the data model (see Figure 1).

The **SPL Tool Developer View** provides a tree-structured representation of the data within a FM consisting of features and groups to closely resemble common graphical representations of FMs on the data level. The non-temporal layer provides classes representing the FM, features, and groups with methods allowing to programmatically modify and query FMs such as `feature.setName(String)` or `feature.getChildGroups()`.

On the temporal layer, an equivalent data structure is employed consisting of temporal features and temporal groups. Due to capturing aspects of evolution, the respective data model is more complex, i.e., model elements potentially subjected to changes are lifted to a first-class entity in the data representation. For instance, as a feature may change its type (e.g., from mandatory to optional) over the course of time, there is a first-class element `FeatureTypeValue` whose instances represent any one of the feature's types along with the individual period of time in which it is valid. The API thus provides useful interfaces for realizing techniques guaranteeing configuration validity during SPL evolution (cf. [26]). In this line, on the temporal layer, the methods to manipulate the respective elements represent evolution operations

that create the adequate structure in the TFM representation, e.g., `temporalFeature.setTypeAt(FeatureType, TemporalPoint)`. Furthermore, the methods querying the elements for information can either be used for a particular point in time to retrieve a single value (e.g., `temporalFeature.getTypesAt(TemporalPoint)`), or for the evolution timeline, in this case delivering a collection of values each carrying information on its temporal validity for a specific period of time (e.g., `temporalFeature.getTypes()`).

The tree-structured data representation within the SPL Tool Developer View allows for intuitive traversal, creation and manipulation of a (T)FM, e.g., within editors or when implementing a FM generator. The different data representations on the non-temporal and temporal layer are sensible when wanting to either abstract from or being fully aware of the intricacies of evolution, e.g., when collaboratively creating a FM or analyzing the development timeline, respectively.

The **Notation Developer View** provides an individual programming interface for both the temporal and non-temporal layer, which comprises all the methods from the SPL Tool Developer View on the respective layer. These interfaces form the basis of our core implementation, which provides essential functionality so that (T)FM developers seeking to integrate with the TFM ecosystem only need to implement a core set of methods (see Section 5.3). Unlike the tree-structured data of the SPL Tool Developer View, which allows for intuitive traversal of the (T)FM and usage of the respective methods, the interfaces in the Notation Developer View loosely resembles a REST-style API that is agnostic of the (T)FMs actual structure, thereby making the implementation of the respective functionality for a specific notation easier. Hence, the interfaces in this view expect the affected model element as the first method parameter, e.g., `setFeatureName(Feature, String)` and `getFeatureChildGroups(Feature)` on the temporal layer, as well as `setFeatureNameAt(Feature, String, TemporalPoint)` and `getFeatureChildGroupsAt(Feature, TemporalPoint)` on the non-temporal layer. The translation to the tree-structured SPL Tool Developer View is performed automatically by the TFM ecosystem, completely transparent for the users of each view.

5.3 Usage and Benefits

The different layers and individual views enable the stereotypical users to use the capabilities of the TFM ecosystem:

The *Notation Developer* can integrate their (T)FM notation on either the non-temporal or temporal layer by implementing a core set of methods of the respective API. We provide an example base implementation that provides essential functionality, such as validating parameters or notifying listeners. This reduces the integration effort to the mere realization of modifications and queries on the underlying native notation, e.g., for a method `setFeatureName(Feature, String)`, a Notation Developer has to implement a companion worker method `setNotationSpecificFeatureName (Feature, String)` performing the change. Capabilities for checking the feature name for validity, notifying of the name change after execution, and updating all representations of the feature including the tree-structured representation of the Notation Developer View are provided automatically by the TFM ecosystem.

The *SPL Tool Developer* can programmatically access various (T)FM notations uniformly via the tree-structured representation to focus on the development of core instead of boilerplate functionality. This reduces the need to deal with format issues and, at the same time, allows implemented tools and analyses to be applicable for a wider range of notations.

The *SPL Manager* creates and manipulates (T)FMs through the provided editors independent of the underlying (T)FM notation. This reduces the intricacies of performing an evolution operation to setting a desired point in time via a slider and performing the editor operation equivalently to a common FM. As a result, the SPL Manager can perform, track, log, analyze and replay evolution for all TFM notations supported by the TFM ecosystem and even for common FMs.

5.4 Capabilities and Requirements Satisfaction

We summarize the essential capabilities of the TFM ecosystem and discuss the satisfaction of the requirements:

Two-layer architecture and temporal adapter. The TFM ecosystem offers both a temporal and a non-temporal layer, allowing transparent use of evolution capabilities via an adapter. The TFM ecosystem offers functionality to build (T)FMs from scratch by starting with an empty model and then adding to it. For both FMs and TFMs, this can be performed explicitly by creating and adding respective elements, e.g., a new feature for an FM or temporal feature for a TFM. This addresses requirements **R1** and **R2**. Regarding **R3** executing the evolution operations provided by the TFM ecosystem not only performs the respective changes on the FM but also creates the TFM elements with appropriate temporal points. These operations can be performed explicitly on the TFM but also implicitly on the FM via the temporal adapter, e.g., by calling `feature.setName(String)` on the temporal adapter. Using the temporal adapter makes it possible to integrate TFM notations seamlessly with tools and analyses intended for common FMs. Modifications via the temporal adapter are performed at one particular temporal point. Queries are performed for a particular temporal point and result in a view on the TFM resembling a common FM.

Views for stereotypical users. We provide a tree-structured representation of (T)FMs as part of the SPL Tool Developer View to satisfy requirement **R4**. Our approach permits intuitive traversal, creation and manipulation, and a representation agnostic of the actual structure of the (T)FM as part of the APIs in the Notation Developer View, which lends itself for implementation. The tree-structured representation

is derived from the API and forwards method calls to the API so that it is synchronized automatically. Furthermore, we defined the APIs in the Notation Developer View on both the non-temporal and temporal level without making assumptions on how individual elements are interconnected, e.g., whether there are multiple groups per feature, just a single group or none at all (as this abstraction can be synthesized by our core implementation). This provides for flexibility with regard to the choice of data source for a (T)FM notation, which includes traditional FM artifacts (e.g., a file in pure::variants) but could also be extended to a database table or a spreadsheet. Upon implementation of the respective API in the Notation Developer View, all abstractions and capabilities of the TFM ecosystem are readily available to the notation. Regarding our goal of harmonizing conceptual differences the abstractions of both the SPL Tool Developer View and Notation Developer View allow for uniform programmatical access to the underlying specific (T)FM notations. Unlike with a format conversion where static artifacts are translated, the abstractions in the TFM ecosystem harmonize individual operations on the specific notations so that translation is performed dynamically with each performed operation. This is an essential prerequisite for future research on using TFMs for collaborative live editing where asynchronous changes have to be integrated dynamically.

Evolution tracking. The TFM ecosystem provides support for listeners that are informed of all applied evolution operations (cf. requirement **R5**). The TFM ecosystem provides interfaces for *non-temporal listeners* and *temporal listeners* which are able to register themselves at the respective layers. Each listener is informed about the state of the involved elements before and after performing the respective evolution operation. The temporal listeners additionally receive the temporal point at which the operation was executed. For instance, the API tracks that a move-feature operation occurred relocating the feature from its old group to a new group at a specific temporal point. This tracking mechanism can be used to undo performed evolution operations as information on the previous state is reported. On basis of this mechanism, the TFM ecosystem allows writing a sequence of changes to a data stream, e.g., to a log file or a database, to document performed evolution operations for tracking and reporting purposes (a prerequisite for requirement **R6**). Furthermore, the TFM ecosystem provides means to record a sequence of changes as a list of Memento [7] objects that capture the respective performed operation and sufficient information on the FM's previous state for reversing its effects (needed for both requirements **R5** and **R6**).

Evolution timelines. The temporal layer of the TFM ecosystem provides full access to FM evolution, thus enabling capabilities for replaying, shifting, and analyzing timelines. The TFM ecosystem allows to detach an evolution history from one specific TFM and replaying it on a different TFM. This functionality is essential for format conversions between different TFM notations as an existing evolution timeline of one notation can be replayed on a newly created TFM of another notation (see requirement **R6**). Based on the evolution recording mechanism, it is also possible to create a TFM out of recorded changes to an FM. This enables to use a FM with its evolution history as a TFM, e.g., to perform evolution analyses which require a TFM as input. The TFM ecosystem offers the option to temporally offset specific evolution operations to shift operation execution forward or backward in time, i.e., the temporal point of the evolution operations does not necessarily have to be the current point in time. This is of particular interest when using TFMs for the planning of future, i.e., not yet performed evolution, where planned evolution operations may have to be postponed (see requirement **R3**). Access to the timeline enables evolution-aware analyses such as inconsistencies between current and planned changes [27] or identifying hotspots of the most frequently changed features (see requirement **R7**).

Access control. With the APIs of the TFM ecosystem on both the non-temporal and temporal layer we present a central access point to control through which all modifications and evolution operations on the FM have to be routed. For example, it is possible to integrate a user management system so that specific rights are required to manipulate portions of a FM. Furthermore, individual evolution operations may be prohibited for portions of a FM within a specific SPL so that, e.g., a core feature may be renamed but may never be deleted (see requirement **R8**).

6 Evaluation

In our evaluation we investigated the expressiveness and utility of our TFM ecosystem.

RQ1. Expressiveness. We assessed to what extent our TFM ecosystem allows to express different concepts of selected subject systems for the purpose of harmonizing their functionality. Specifically, we investigated three questions in this regard: Which functionality could be covered by our API? Which difficulties did we experience, e.g., which specifics of these systems could not be mapped (thus indicating gaps in our solution)? Which effort was needed for the implementation? We implemented the API interfaces for two of the TFM subject approaches (DarwinSPL and FORCE) and three FM modeling approaches (DeltaEcore, pure::variants, FeatureIDE) to investigate to what extent the operations can be covered, and to better understand potential deficiencies of our solution.

RQ2. Utility. We investigated the ability of our TFM ecosystem to satisfy the needs of developers in concrete TFM scenarios. Specifically, we tested our API based on existing evolution histories of two systems to answer the following question: Do the results of the evolution scenarios replayed

Table 2. Lines of API integration code and API calls needed to implement the TFM API operation for the selected tools.

TFM Operation	DarwinSPL		FORCE		DeltaEcore		FeatureIDE		pure::variants	
	Code	API	Code	API	Code	API	Code	API	Code	API
Create Model	16	14	17	13	4	3	4	7	9	5
Create Feature	8	7	3	2	3	2	1	2	17	5
Create Group	3	3	3	N/A	2	2	4	N/A	5	N/A
Add Feature	3	2	15	2	20	3	9	2	29	3
Add Group	3	2	17	2	2	2	11	3	43	3
Move Feature	6	4	19	2	37	7	11	5	49	6
Move Group	6	4	19	2	5	5	24	6	47	3
Remove Feature	4	4	1	1	17	4	1	1	23	3
Remove Group	3	2	3	1	3	3	6	1	3	N/A
Rename Feature	3	2	1	1	1	1	1	1	17	4
Change Feature Variability Type	17	2	16	4	10	1	14	2	23	4
Change Group Variability Type	20	2	26	7	14	1	22	1	19	3

for one of the TFM notations match the results of other TFM notations replayed using our API?

6.1 RQ1. Expressiveness

Concerning the selection of the TFM tools, our goal was to include as many as possible in our evaluation. However, we had to exclude TFM approaches with no publicly available implementation (FDV, EvoFM). We further excluded SuperMod because revisions are only supported internally but cannot be accessed through its API. Furthermore, DeltaEcore is the basis for DarwinSPL, so we evaluated only the more recent approach as described in our research method. Regarding FM tools, our main criterion was to include widely used tools in our evaluation. We thus included one popular commercial tool (pure::variants), one popular academic/OSS tool (FeatureIDE), and an additional tool we had access to (DeltaEcore's non-temporal layer). Overall, for demonstrating the expressiveness of our solution, we integrated five different tools via our Notation Developer View reference implementation, thus showing that it can cover the common operations of various tool APIs.

As a metric for the implementation effort of the (T)FM API, we present in Table 2 the number of source lines of code (SLOC) required to implement the specified TFM API operation for the given (T)FM approaches or tools. Additionally, we present the number of tool API operations, which were called to implement our TFM API operations for the specific notations.

DarwinSPL. The implementation of the API for DarwinSPL captures all functionality of the temporal part of the API. As the structure of the temporal API is similar to DarwinSPL's TFM itself, we did not encounter significant challenges. For the modification operations, we could directly reuse evolution operations provided by DarwinSPL. This is also represented in Table 2 as the number of SLOC and API Operations are not very different in most cases, indicating that the TFM operations could be directly forwarded to the corresponding DarwinSPL API operations. The only major difference is that DarwinSPL does not have the concept of a default group which is supported by the API. Consequently, we had to define which of the child groups is considered the default group. Compared to the API, DarwinSPL provides additional functionality that is currently not represented on API-level. For instance, the mapping to implementation artifacts and their evolution are not yet considered in the API. Moreover, DarwinSPL supports feature attributes, which are also not available in the API.

FORCE. Overall, our implementation demonstrated that our API covers the FM modification operations of FORCE. However, we also experienced a number of challenges. In contrast to other TFMs, the FM modification operations of FORCE do not support specifying temporal points at which the operations are taking place. This is because FORCE is part of a feature-based engineering platform and variation control system. The key application scenario of FORCE is to track the evolution of FMs in context with a series of commits and respective revision numbers of features. This means, the FMs are modified implicitly at the current temporal point while the FORCE API operations appear as non-temporal operations. However, a temporal context is required to retrieve a specific FM. This behavior is very specific to FORCE and thus not part of the common TFM API. However, we were still able to develop an API interface for FORCE allowing to use the non-temporal common API calls for modification operations and the temporal operations for FM retrieval. An additional problem was that FORCE does not provide the concept of a group as a distinct entity. Groups thus needed to be handled by managing surrogate groups in the API. This resulted in additional code overhead mostly present in the implementation of the `add` and `move` operations, which required 15 to 19 SLOC but only two FORCE API calls (Table 2). FORCE allows to divide large FMs into an interrelated set of

smaller FMs via components to deal with model complexity. This capability is specific to FORCE and could thus not be mapped to operations of the API.

DeltaEcore. DeltaEcore's support for evolution mostly concerns the solution space whereas its Hyper Feature Models (HFMs) allow for variability modeling with versions but do not capture changes to the structure of the FM. Hence, our implementation covers the non-temporal layer of the API allowing for variability modeling without evolution. As HFMs lack a concept for a default group, we synthesized the respective behavior intended by the API by designating the first group as the default group. The increased number of SLOC required to implement `add` and `move` operations is required to set the variation type (AND, OR, XOR). It was not possible to represent the versions of a feature's implementation of HFMs due to the lack of a similar concept in the API reducing HFMs to common FMs. At the same time, the integration of HFMs with the non-temporal API, together with the provided temporal adapter, extends the capacity of HFMs as it is now also possible to capture evolutionary changes to the structure of an HFM.

FeatureIDE. This tool does not support evolution, so only the non-temporal FM part of our API was relevant. Our API covers the entire FeatureIDE functionality. In fact, most challenges were caused by restricting additional functionality of our API to the comparatively limited functionality of FeatureIDE. For example, FeatureIDE allows only one group below a feature while our API allows an arbitrary number of groups. We thus had to ensure that not more than one group per feature is added via our API. Additionally, similar to FORCE, FeatureIDE does not treat groups as own entities as our TFM API does. Therefore, we again had to create and maintain surrogate objects for groups when implementing the API. Again, similar to FORCE, the major code overhead is created for the `add` and `move` operations (Table 2). We could not cover mandatory features in OR and ALTERNATIVE groups (a capability of FeatureIDE) using our API.

pure::variants. Similar to FeatureIDE, pure:variants does not support evolution via temporal operations. Nonetheless, we were able to provide a fully functional implementation of the non-temporal API for this commercial tool. Again, similar to FORCE and FeatureIDE, the main challenge was the adaptation of the group concept as pure::variants does not explicitly model groups. As a consequence, surrogate groups had to be implemented. Conceptually, it is possible to have up to three child groups under one feature, i.e., one group per group type (AND, OR, ALTERNATIVE). In pure::variants, a feature may have the type OPTIONAL, MANDATORY, OR or ALTERNATIVE. OPTIONAL and MANDATORY features are implicitly contained in an AND group. OR and ALTERNATIVE features are implicitly contained in groups with the same type. Consequently, we had to ensure that at most one group per type is created under each feature. Moreover,

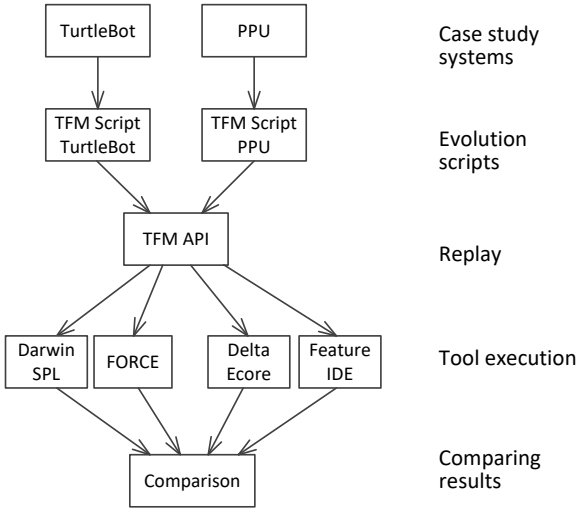

Figure 3. Research method for RQ2 (utility).

features created in pure::variants have to be directly added in the FM hierarchy, while features are first created and then added to the hierarchy in our API. To support the API operations, the status of features has to be managed manually, i.e., created, added, detached. As a consequence of these limitations, the source code overhead was significantly higher than for FORCE or FeatureIDE. If looking at Table 2, the implementation of API operations for pure::variants required between 17 to 49 SLOC of which just 3 to 6 are pure::variants API calls.

6.2 RQ2. Utility

We tested our TFM ecosystem based on existing evolution histories of two academic case study systems to demonstrate that the API allows replaying evolution scenarios in different tool implementations. Specifically, we replayed evolution scenarios developed for one of the TFM notations in multiple other TFM notations by utilizing our API. Our overall approach is summarized in Figure 3. Two authors of the paper analyzed existing evolution histories of the case study systems TurtleBot and Pick-and-Place Unit (PPU) and created scripts containing the found change operations as operations of our TFM API. The resulting evolution scripts were then executed to replay the evolution scenarios with four different tools: DarwinSPL, FORCE, DeltaEcore, and FeatureIDE. Finally, we compared the results obtained from the replays.

TurtleBot [38, 39] is a small domestic robot used for collecting and delivering goods, e.g., it can be loaded with remotely requested medical goods at a dispenser and delivers them to a patient and their doctor. TurtleBot was previously used by DeltaEcore as a case study system for validating notations and tools. The driver for TurtleBot is structured in the sense of an SPL with various features regarding the robot's means of locomotion and control of its movement. It

is possible to generate custom-tailored variants of the driver to accommodate for the different existing configurations of the robot while respecting the individual robots' limited resources with respect to CPU and battery life. The driver with its respective FM and implementation artifacts has been developed in multiple iterations where individual features have been added, modified or removed over the course of evolution. The final version of TurtleBot consists of 11 features including optional features and alternatives.

Pick-and-Place Unit (PPU) [15] is a manufacturing system for transporting and sorting different work pieces and a well-known example from the automation domain. Based on the PPU evolution history described in [41] a developer (not an author of this paper) implemented a PPU product line with a set of different revisions and variants representing different development stages. The PPU systems consist of different subsystems, i.e., a control part and a visualization part, which are implemented in two different programming languages, an IEC-61131-3 compliant language for the control components and Java for the user interface. This strongly conforms to the architecture prevalent in the domain. The PPU evolution history contains 14 different revisions in total. The FM of the first version has 10 features, while the FM of the last and most complex version has 26 features. The different versions again represent configurable systems with alternative and optional features from which concrete systems can be derived.

Results. Executing the script of both evolution cases with each of the four API implementations resulted in four FMs for the PPU and four for TurtleBot. Each FM was the result of using our common API but different (T)FM notations and tools in the background. We manually inspected the resulting FMs for PPU and TurtleBot to compare them. We found that applying our common (T)FM API always yielded identical FMs, regardless of which (T)FM notation was used to implement it. In that way we showed utility of the TFM API for re-creating a change history created in one TFM tool environment in another one.

6.3 Threats to Validity

As with any evaluation our results may not generalize beyond the cases we chose for our evaluation. However, to mitigate that bias we included both representative TFM approaches from research and multiple FM tools (including a commercial one) when developing the API interfaces. Moreover, we avoided generalizations and presented detailed experiences of implementing the tool interfaces. There is also a potential bias caused by the selection of the case study systems TurtleBot and PPU. However, the PPU system [41] is regarded as a standard example and considered representative for the domain. Furthermore, the size of the case studies is rather small. However, for the purpose of this evaluation we needed systems for which an evolution history is available for both the FM and the implementation, thus significantly reducing the available options. Furthermore, the main foci of the evaluation were to assess expressiveness and utility, for which the selected case study systems were sufficient. It could further be argued that some authors of this paper made significant contributions when creating the evolution history of the FMs. However, for the PPU, we followed the descriptions and models as found in the literature. Additionally, the implementation and FMs were created by a developer not part of the author team.

7 Conclusion and Future Work

We presented a TFM ecosystem which aims at harmonizing existing (T)FM notations and providing common capabilities to perform, track, analyze and replay changes on (temporal) FMs. Our work is based on a comparison of six TFM approaches carried out to understand their commonalities and specific capabilities. The achieved harmonization benefits SPL tool developers who can utilize the common TFM API for implementations enabling different TFM notations to be usable in the back-end. Harmonization further benefits TFM notation creators who can integrate into an existing set of tools, even with a newly devised notation, thus reducing the need for re-creating boilerplate functionality. We regard the abstraction via the API as superior to an interchange format, as no details are lost during conversion. We demonstrated the approach by providing implementations for existing TFM notations and tested its feasibility by replaying evolution histories of two SPLs. We provide our work as a common foundation for future research in the area of TFMs. Overall, our work presented clears significant technical obstacles to foster collaboration and enables future research directions in the area of SPL evolution using TFMs.

In our future work we will complement our TFM ecosystem based on our experiences and lessons learned. Specifically, we plan to provide capabilities supporting the evolution of implementation-to-artifact mappings. We will further provide capabilities for branching and merging to synchronize asynchronously performed changes, based on our ongoing efforts to create such support for our own subject approaches. Finally, the TFM ecosystem provides a good starting point to investigate support for querying and modifying of intervals of the evolution timeline.

Acknowledgments

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and KEBA AG, Austria is gratefully acknowledged. This work was partially supported by the Federal Ministry of Education and Research of Germany within CrESt (funding 01IS16043S).

References

[1] 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.

[2] Goetz Botterweck, Andreas Pleuss, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. 2010. EvoFM: Feature-driven Planning of Product-line Evolution. In *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering (PLEASE)*. ACM, New York, NY, USA, 24–31.

[3] Goetz Botterweck, Andreas Pleuss, Andreas Polzer, and Stefan Kowalewski. 2009. Towards Feature-driven Planning of Product-line Evolution. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 109–116.

[4] David Budgen, Mark Turner, Pearl Brereton, and Barbara Kitchenham. 2008. Using mapping studies in software engineering. In *Proceedings of the 20th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, Vol. 8. 195–204.

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

[6] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[8] Karine Gomes, Leopoldo Teixeira, Thayonara Alves, Márcio Ribeiro, and Rohit Gheyi. 2019. Characterizing Safe and Partially Safe Evolution Scenarios in Product Lines: An Empirical Study. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS)*. ACM, 15:1–15:9.

[9] Øystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. 2012. CVL: common variability language. In *Proceedings of the 16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7*. ACM, 266–267.

[10] Wolfgang Heider, Roman Froschauer, Paul Grünbacher, Rick Rabiser, and Deepak Dhungana. 2010. Simulating evolution in model-based product line engineering. *Information & Software Technology* 52, 7 (2010), 758–769.

[11] Wolfgang Heider, Rick Rabiser, and Paul Grünbacher. 2012. Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. *Software Tools for Technology Transfer* 14, 5 (2012), 613–630.

[12] Robert Hellebrand, Adeline Silva, Martin Becker, Bo Zhang, Krzysztof Sierszecki, and Juha Savolainen. 2014. Coevolution of Variability Models and Code: An Industrial Case Study. In *Proceedings of the 18th International Software Product Line Conference - Volume 1 (SPLC '14)*. ACM, New York, NY, USA, 274–283.

[13] Daniel Hinterreiter. 2018. Supporting Feature-oriented Development and Evolution in Industrial Software Ecosystems. In *Proceedings of the 22nd International Systems and Software Product Line Conference – Volume 2 (SPLC)*. ACM, 79–86.

[14] Daniel Hinterreiter, Kevin Feichtinger, Lukas Linsbauer, Herbert Prähofer, and Paul Grünbacher. 2019. Supporting Feature Model Evolution by Lifting Code-Level Dependencies: A Research Preview. In *Proceedings of the Conference on Requirements Engineering: Foundation for Software Quality*. Springer International Publishing, 169–175.

[15] Daniel Hinterreiter, Herbert Prähofer, Lukas Linsbauer, Paul Grünbacher, Florian Reisinger, and Alexander Egyed. 2018. Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems. In *Proceedings 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, Turin, 107–114.

[16] Jean-Marc Jézéquel. 2012. Model-driven engineering for software product lines. *ISRN Software Engineering* (2012).

[17] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proceedings of the 16th International Conference on Generative Programming: Concepts & Experience (GPCE)*. ACM, 49–62.

[18] Tomi Männistö and Reijo Sulonen. 1999. Evolution of Schema and Individuals of Configurable Products. In *Advances in Conceptual Modeling*, Peter P. Chen, David W. Embley, Jacques Kouloumdjian, Stephen W. Liddle, and John F. Roddick (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 12–23.

[19] Maira Marques, Jocelyn Simmonds, Pedro O. Rossel, and María Cecilia Bastarrica. 2019. Software product line evolution: A systematic literature review. *Information & Software Technology* 105 (2019), 190–208.

[20] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2018. Context-aware reconfiguration in evolving software product lines. *Science of Computer Programming* 163 (2018), 139–159.

[21] Ralf Mitschke and Michael Eichberg. 2008. Supporting the Evolution of Software Product Lines. In *ECMDA Traceability Workshop (ECMA-TW)*. SINTEF, Norway, 87–96.

[22] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa, and Paulo Borba. 2011. Investigating the Safe Evolution of Software Product Lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 33–42.

[23] Robert C. Nickerson, Upkar Varshney, and Jan Muntermann. 2013. A method for taxonomy development and its application in information systems. *European Journal of Information Systems* 22, 3 (2013), 336–359.

[24] 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)*. ACM, New York, NY, USA, 92–99.

[25] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-model Evolution. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 188–201.

[26] 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)*. ACM, 73–80.

[27] Michael Nieke, Christoph Seidl, and Thomas Thüm. 2018. Back to the Future: Avoiding Paradoxes in Feature-model Evolution. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 2 (SPLC)*. ACM, 48–51.

[28] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrezj Wasowski, Christian Kästner, Jianmei Guo, and Claus Hunsen. 2013. Feature-Oriented Software Evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*. ACM.

[29] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wasowski, and Paulo Borba. 2013. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *17th International Software Product Line Conference*. ACM.

[30] Juliana Alves Pereira, Kattiana Constantino, and Eduardo Figueiredo. 2015. A Systematic Literature Review of Software Product Line Management Tools. In *Proceedings of the 14th International Conference on Software Reuse (ICSR) (Lecture Notes in Computer Science)*, Ina Schaefer and Ioannis Stamelos (Eds.), Vol. 8919. Springer, 73–89.

[31] Daniela Rabiser, Herbert Prähofer, Paul Grünbacher, Michael Petruzelka, Klaus Eder, Florian Angerer, Mario Kromoser, and Andreas Grimmer. 2018. Multi-purpose, multi-level feature modeling of

large-scale industrial software systems. *Software & Systems Modeling* 17, 3 (July 2018), 913–938.

[32] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Computer Networks* 51, 2 (2007), 456–479.

[33] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. 2016. Aligning coevolving artifacts between software product lines and products. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 9–16.

[34] Felix Schwägerl. 2018. *Version Control and Product Lines in Model-Driven Software Engineering*. Ph.D. Dissertation. University of Bayreuth, Germany.

[35] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: Tool support for collaborative filtered model-driven software product line engineering. In *Proceedings 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 822–827.

[36] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. 2012. Co-Evolution of Models and Feature Mapping in Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*.

[37] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore – A Model-Based Delta Language Generation Framework. In *Modellierung*, Vol. 19. Gesellschaft für Informatik e.V., Bonn, 21.

[38] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Integrated Management of Variability in Space and Time in Software Families. In *Proceedings of the 18th International Software Product Line Conference (SPLC)*. ACM, 22–31.

[39] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2013. 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)*. ACM, 6:1–6:8.

[40] Christoph Seidl, Tim Winkelmann, and Ina Schaefer. 2016. A software product line of feature modeling notations and cross-tree constraint languages. In *Modellierung 2016*, Andreas Oberweis and Ralf Reussner (Eds.). Gesellschaft für Informatik e.V., Bonn, 157–172.

[41] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. 2014. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit*. Technical Report. Technische Universität München.