Anomaly Analyses for Feature-Model Evolution

Anonymous Author(s)

Abstract

Software Product Lines (SPLs) are a common technique to capture families of software products in terms of commonalities and variabilities. On a conceptual level, functionality of an SPL is modeled in terms of features in Feature Models (FMs). As other software systems, SPLs and their FMs are subject to evolution that may lead to the introduction of anomalies (e.g., non-selectable features). To fix such anomalies, developers need to understand the cause for them. However, for large evolution histories and large SPLs, explanations may become very long and, as a consequence, hard to understand. In this paper, we present a novel method for anomaly detection and explanation that, by encoding the entire evolution history, identifies the evolution step of anomaly introduction and explains which of the performed evolution operations lead to it. In our evaluation, we show that our method significantly reduces the complexity of generated explanations.

CCS Concepts • **Software and its engineering → Software product lines**; **Software evolution**;

Keywords Software Product Line, Feature Model, Evolution, Anomalies, Explanation, Evolution Operation

ACM Reference Format:
Anonymous Author(s). 2018. Anomaly Analyses for Feature-Model Evolution. In *Proceedings of 17th International Conference on Generative Programming: Concepts & Experience (GPCE'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.475/123_4

1 Introduction

A *Software Product Line (SPL)* is an approach to manage reuse for families of software products [38, 43]. Functionality of the products of an SPL is captured on a conceptual level using *features*. Commonly, features are organized in a *Feature Model (FM)*, expressing relations between features in a tree-like notation [15] and *cross-tree constraints* [16]. A *configuration* is a set of selected features and it is valid if the feature selection satisfies all constraints posed by the FM and the cross-tree constraints. A valid configuration is used to derive *products* of an SPL using variability realization mechanisms [43].

To satisfy software requirement changes, SPLs need to evolve [24, 36]. As Passos et al. motivated, this evolution optimally starts by evolving the FM [35]. When defining and changing feature models, anomalies may be introduced inadvertently, e.g., preventing the creation of valid configurations (void FM anomaly) or the selection of certain features [3]. Fixing anomalies is a very complex task and, thus, entails significant costs. Understanding the cause for anomalies is a complex but crucial activity to be able to fix them.

Several approaches exist to detect [14, 51] and explain [2, 9, 10, 17, 18, 20, 26, 41, 49, 50] FM anomalies in terms of violated constraints. For large FMs, some explanations have a significant size as a large percentage of features and cross-tree constraints contribute to the corresponding anomaly, making it hard to understand them. For instance, a recent bug in the tool FeatureIDE resulted in few group types of FMs to be changed.[1] For a large FM from industry (712 features and 1141 cross-constraints), this resulted in three group types to be changed and as a consequence a void FM anomaly occurred. The explanation of this anomaly contained 91 constraints and 92 features were involved. Thus, developers had to inspect all these constraints and features despite the immediate cause being the change of the three group types.

Over the course of time, SPLs may have long evolution histories. As evolution yields additional complexity, the likelihood that developers inadvertently introduce anomalies during this evolution increases. Without proper methods for detection, the introduced anomalies remain undetected, thus, harming the FM consistency. Hence, detecting and fixing these anomalies not just retroactively, after they caused harm, but proactively during maintenance is well advised. For this reason, developers need to identify the anomalies in the entire evolution history and pinpoint the FM version of the anomaly introduction. With current approaches, this requires significant manual effort as each evolution step needs to be analyzed on its own and all anomalies have to be traced manually throughout the entire history.

When planning future FM evolution, developers might draft several evolution steps FM [33], each possibly consisting of several evolution operations applied to the FM. Due

GPCE'18, November 2018, Boston, MA, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

[1] https://github.com/FeatureIDE/FeatureIDE/issues/662

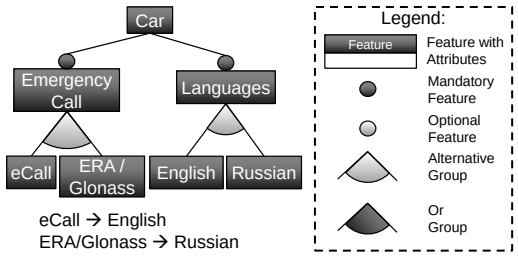


Figure 1. FM of an in-Car Emergency Call System

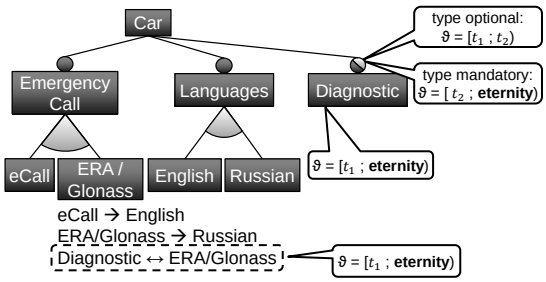


Figure 2. Evolution of the Running Example as TFM

to the complexity of anomaly detection and fixing, it is infeasible for developers to analyze the FM after each evolution operation. Instead, multiple evolution operations have been performed before searching for anomalies. Moreover, when fixing anomalies, developers might need to perform multiple operations which may introduce additional anomalies. If developers introduced anomalies in such planned evolution steps, it is hard to understand which of their operations lead to the anomaly. This is particularly important if few evolution operations result in large explanations. Existing approaches are not able to explain which of the operations performed by the developers lead to an anomaly.

We overcome these limitations by incorporating FM evolution information for anomaly detection and explanations. In particular, we make the following contributions:

- We propose a method allowing the detection of all anomalies in the evolution history and pinpointing the evolution step of introduction by analyzing the FM evolution in its entirety (as opposed to evolution steps individually).
- We introduce a novel concept for explaining anomalies by identifying causing evolution operations and, thus, reducing explanation complexity.
- We provide tool support for anomaly detection, explanation generation and their inspection.
- We evaluate our method by analyzing evolution histories of three real-world FMs, showing correctness, measuring performance and explanation reduction rates.

With these contributions, we are able to efficiently detect all anomalies in the past and future evolution of SPLs and, most importantly, explain them in a less complex way.

2 Background

A Feature Model (FM) is the most common representation of variability in an SPL on conceptual level [15, 43]. Features of an FM are structured in a tree-like notation. As an example, Figure 1 shows an FM of an in-car emergency call system. Two features representing different implementations of emergency call systems are available, i.e., `eCall` and `ERA/Glonass`. Moreover, two different language features are available, i.e., `English` for Europe and `Russian` for Russia. Each feature can only be selected if its parent is selected. A feature can be either *mandatory* (e.g., feature `Languages`) and must be selected if its parent in the feature tree is selected, or *optional*. Additionally, features may be grouped: An *Or* group states that at least one of the group's features has to be selected if the parent feature is selected while an *Alternative* group states that exactly one of the group's features has to be selected if the parent feature is selected.

To allow developers to model evolution of FMs, Nieke et al. devised the general concept of *temporal elements* [32]. Each temporal element has a *temporal validity* $\vartheta = [\vartheta_{since}; \vartheta_{until})$ – a right-open interval of dates stating the timespan in which the element is valid. Using temporal elements, it is not only possible to model past evolution history but also plan future evolution [33]. Nieke et al. applied this concept to FMs, resulting in *Temporal Feature Models (TFMs)* [32]. In TFMs, each element of the FM is modeled as temporal element. Thus, all features, names, and groups are modeled as temporal elements. As feature and group types need to evolve, they are modeled as temporal elements as well. To move features in the feature tree, the relations between features and their groups need to be modeled as temporal elements and, thus, as own entity.

The definition of the temporal validity allows to seamlessly change elements, such as a feature type, in the evolution history. As the temporal validity is a right-open interval, a change of a feature type from $type_{old}$ to $type_{new}$ at point in time t can be realized by setting the end of the temporal validity of the old feature type to the beginning of the temporal validity of the new feature type: $\vartheta_{type_{old}} = [eternity; t)$, $\vartheta_{type_{new}} = [t; eternity)$. Hence, we are able to model arbitrary FM evolution with TFMs and to capture its entire evolution history in one model. Each unique date contained in all temporal validities of a model represents one evolution step. As the dates of temporal validities are not limited to past dates but may also be in the future, TFMs also facilitate future planning of FM evolution [33].

Figure 2 shows the evolution of the running example modeled in a TFM. The temporal validities are illustrated as annotation at the model. In this TFM, three versions and two evolution steps are modeled: The first version at t_0, the second version at t_1 and the third version at t_2. In this example, a new feature and a new cross-tree constraint are introduced at t_1. As the type of the new feature is *optional*, it is added to

the feature as well. Thus, the temporal validities of the new feature, its *optional* type and the new cross-tree constraint are set to $\vartheta = [\vartheta_{since} = t_1; eternity)$. As the type of the new feature is changed at t_2 from *optional* to *mandatory*, a new respective *mandatory* feature type element is added. To replace the *optional* feature type, the temporal validity of the *optional* feature type is changed to $\vartheta = [\vartheta_{since} = t_1; \vartheta_{until} = t_2)$ and the temporal validity of the *mandatory* feature type is set to $\vartheta = [\vartheta_{since} = t_2; eternity)$. Nieke et al. implemented TFMs and respective editors in a tool suite . The this tool suite editors hide the complexity of TFMs by allowing developers to apply changes to the FM, as common in other editors, which are translated to temporal elements in the background [31].

Anomalies in FMs describe design flaws and mismodeling of FMs. Benavides et al. identified a set of relevant anomalies developers should avoid [3]. In particular, they introduced three main types of anomalies: i) *void FM* anomaly if no valid configuration of the FM exists; ii) *dead* feature anomaly if a feature cannot be selected in any valid configuration; iii) *false-optional* feature anomaly if a non-mandatory feature is part of each valid configuration. For instance, in the running example, at t_2, the feature `ERA/Glonass` is in an *alternative* group. However, as feature `Diagnostic` is a *mandatory* feature and the last constraint defines that `ERA/Glonass` and `Diagnostic` must always be selected together, `ERA/Glonass` became *false-optional*. As a consequence, the feature `eCall` became *dead* as it is in an *alternative* group with `ERA/Glonass`.

3 Evolution-Aware Anomaly Detection

Several approaches exist to detect [14, 51] and explain [2, 9, 10, 17, 18, 20, 26, 41, 49, 50] FM anomalies in terms of violated constraints. An explanation of an anomaly consists of the set of constraints that cannot be satisfied altogether and, thus, lead to the anomaly. To the best of our knowledge, none of the existing methods is considering evolution. Thus, when analyzing the evolution history of an FM, the entire analysis has to be performed for each evolution step on its own. Additionally, developers have to search manually for the evolution step in which an anomaly has been introduced, which can be hard or unfeasible for large evolution histories. Even more important, when performing FM evolution, anomalies may be introduced by a small set of evolution operations but the explanation using current methods may become extremely large (cf. example in the Introduction with more than 90 constraints involved). Thus, the existing methods do not provide any information to developers on which of their performed evolution operations caused an anomaly. As a consequence, fixing anomalies becomes complex both for existing and upcoming evolution steps and because of too much complexity, anomalies might not be fixed.

When searching for anomalies, existing approaches construct satisfiability problems and solve them by querying of-the-shelf solvers. In these satisfiability problems, all features are translated into variables that can be either *true* (i.e., selected) or *false* (i.e., deselected). The constraints imposed by the FM structure and the cross-tree constraints are translated into a set of formulas. For instance, Listing 1 shows the propositional formulas for the TFM of the running example and its cross-tree constraints at t_0. Many of those formulas remain the same for each evolution step. For instance, for the running example, all formulas of t_0 remain stable for the whole evolution history as only new formulas are added. This results in redundancies in queries for solvers if multiple evolution steps are analyzed.

```
1  Car
2  Car → (EmergencyCall ∧ Languages)
3  (EmergencyCall ∨ Languages) → Car
4  EmergencyCall → (eCall ⊕ ERA/Glonass)
5  (eCall ∨ ERA/Glonass) → EmergencyCall
6  Languages → (English ⊕ Russian)
7  (English ∨ Russian) → Languages
8  eCall → English
9  ERA/Glonass → Russian
```

Listing 1. Propositional Formulas of the Running Example at t_0 ($\oplus$ stands for the exclusive or operator).

The idea of the evolution-aware anomaly detection is to incorporate anomaly detection and explanation with FM evolution by encoding the entire FM evolution history in one set of variables and one set of formulas for a solver. This way, it is possible to i) reuse the solver for formulas that remain stable over multiple evolution steps and the entire evolution history is analyzed automatically; ii) detect the evolution step at which an anomaly first arose; iii) derive evolution operations performed by developers to reduce explanation complexity.

3.1 Encoding the Evolution History

In this paper, we incorporate FM evolution for anomaly detection and explanation by utilizing TFMs. Instead of having individual states of an evolved FM, a TFM provides additional information about the evolution, i.e., model elements that changed and the corresponding evolution operations (cf. Section 2). To capitalize on this fact and, thereby, reduce the number of requests to the solver, we encode the notion of change over passing time into one single request. For this purpose, we *tag* evolving formulas with the time spans for which they are valid. Formulas that remain the same for the entire evolution history can be left as they are. These tags tell the solver for which point in time it needs to consider the tagged formula. To derive which formulas need to be tagged, we make use of the *temporal elements* of TFMs (cf. Section 2). Each formula is generated from a certain structure of the FM tree, such as group compositions or feature types, and these structures are modeled as temporal elements. As each temporal element has a temporal validity (i.e., the interval

of dates $[\vartheta_{since}; \vartheta_{until})$ at which it is temporally valid), we can generate these tags by using this information. A tagged formula looks as follows:

$$(\text{evolution} \geq t_i \wedge \text{evolution} < t_j) \rightarrow (\text{original formula})$$

To enable such an encoding and to be able to solve such formulas (e.g., using an SMT solver) we introduce a new variable representing the evolution steps. As we may have multiple evolution steps, this variable has to have a larger domain than just *true* and *false* and, as a consequence, propositional formulas are not enough. Thus in the following, we define formulas in a first-order style notation which allows us to also use variables having an Integer domain. As a variable representing the evolution steps, we need to identify all relevant steps. For this purpose, we make again use of the temporal elements of TFMs. We can identify all relevant steps by inspecting the temporal validities of all temporal elements of a TFM. To represent the identified evolution in tagged formulas, the newly introduced variable is an Integer, denoted in the following as *evolution variable*, to represent these evolution steps. The domain of this variable is $[0; n]$, whereas n is the number of evolution steps identified in the TFM (e.g., $n = 2$ for the running example). The value 0 for the evolution variable represents the state before the first evolution step (e.g., the TFM at t_0 of the running example in Figure 2).

Using this evolution variable, the tagged constraints can be generated. Listing 2 shows the set of formulas for the entire evolution history of the running example. As can be seen, in this case, only three formulas need to be tagged and the rest can be reused for each evolution step. This way, a solver only needs to evaluate the original formula if the value of the evolution variable lies within the defined interval.

```
1  Car
2  Car → EmergencyCall ∧ Languages
3  EmergencyCall ∨ Languages → Car
4  EmergencyCall → eCall ⊕ ERA/Glonass
5  eCall ∨ ERA/Glonass → EmergencyCall
6  Languages → English ⊕ Russian
7  English ∨ Russian → Languages
8  eCall → English
9  ERA/Glonass → Russian
10 (evolution ≥ 0) → (Diagnostic → Car)
11 (evolution ≥ 0) → (Diagnostic ↔ ERA/
       Glonass)
12 (evolution ≥ 1) → (Car → Diagnostic)
```

Listing 2. Formulas of the Entire Evolution History of the Running Example

Note that our encoding allows seamlessly three types of analyses: past evolution history, currently performed evolution, and already pre-planned evolution steps. After this translation and tagging, the formulas can be used as basis to detect anomalies using of-the-shelf solvers. In the following, we explain how we construct formulas to search for anomalies in the entire evolution history.

3.2 Solving TFM Evolution Histories

Existing methods to solve queries for FM analyses are not aware of evolution and, thus, do not incorporate it. To overcome this limitation, we present a method to generate queries on TFM evolution for SMT solvers.

To understand how we detect anomalies in the evolution history, let us assume that, given a conjunction of constraints TFM_c that defines a TFM, we denote the set of all features as F, attributes as A, and the evolution variable as e.

To check whether a TFM is valid at a given time t, it is possible to check whether $SAT((e = t) \wedge TFM_c)$. If this formula is satisfiable, this means that for the time t (we require the evolution variable to be equal to t by enforcing $e = t$) there is a possible set of feature in F that represents a valid configuration. Conversely, if the formula is unsatisfiable, no possible selection of feature of F will allow all the constraints to be satisfied, thus, implying the that the TFM is void.

To check whether a feature $f \in F$ is a dead feature at time t, it is possible to check whether $SAT(f \wedge (e = t) \wedge TFM_c)$. If this formula is satisfiable, this means that for the time t all the constraints that defined the TFM are satisfied. Moreover, also the feature f can be selected, thus, implying that f is not a dead feature for the time t. Conversely, if the previous formula is unsatisfiable, no possible assignment to features will allow feature f to be selected and, thus, proving that f is dead.

The check to determine if a feature is false optional is very similar. While in the previous case we enforced the feature f to be selected, in this case we enforce the feature to be deselected by checking the formula $SAT(\neg f \wedge (e = t) \wedge TFM_c)$. Similar as for the dead feature check, if this formula is satisfied, there exists a possible valid configuration in which the feature f is deselected proving that f is not false optional.

To find all dead and false-optional features in the evolution history, we need to know which feature to check at which evolution step. For this purpose, we introduce a list `toCheck` being a list of pairs of features and a list of values for the evolution variable for which the feature is analyzed. As mandatory features may not be false-optional and a dead mandatory features imply either that their parent is dead as well or a void FM anomaly, we add all features features with all dates at which they are not mandatory to the list `toCheck`. As feature types are modeled as temporal elements in TFMs, we iterate over all feature types of a feature and generate the list of values for the evolution variable by using the temporal validities of optional types.

For checking dead and false-optional features, we iteratively try to satisfy the previously mentioned formulas for all the optional features and time points in `toCheck`. Similarly, the validity check is performed for all the possible time

points. Due to the fact that the encoding of the entire evolution history is done in just one formula, what is learned trying to satisfy a previous formula can be reused to find other dead or false-optional features or prove the not validity of a TFM faster. For this purpose, incremental solvers can be used that allow the addition of constraints on-the-fly without restarting the search from scratch. In particular, for the dead feature analysis, we first check if TFM_c is satisfiable. Then we iterates over all the possible $\langle f, t \rangle \in \texttt{toCheck}$. For every pair, we incrementally set the value of the evolution variable to t and set the feature f to true. Then, we check whether the formula is unsatisfiable. If that is true, then f is a dead feature for the point in time t. A similar check can be performed for false-optional features. After detecting the anomalies using the previously introduced method, we still want to identify which evolution operations performed by developers caused that anomaly which we will discuss in the following.

4 Evolution-Aware Anomaly Explanation

When developers change FMs during evolution, this may result in many operations applied to the FM – especially for planning of future SPL evolution. Understanding FM anomalies in order to fix them can be a very complex task and the number of anomalies can increase very quickly as can be seen in the running example. As help for developers to understand anomalies, several methods exist to explain anomalies in terms of constraints that cannot be fulfilled [2, 9, 10, 14, 17, 18, 20, 26, 41, 49, 50]. However, for large FMs, anomaly explanation length grows quickly as many features and constraints may be involved and during evolution, a single evolution operation can cause an anomaly with a large explanation. When developers try to fix such anomalies using existing methods, in the worst case, they have to study the entire explanation to identify the cause of the anomaly. The existing methods do not incorporate information about evolution and, thus, are not able to identify evolution operations that caused a particular anomaly.

To overcome the previously mentioned limitations we explicitly incorporate information on the FM evolution in two ways. First, we explain anomalies for the point in time they were introduced taking over the task of searching for the correct point in time to fix anomalies. Second, we identify evolution operations performed by developers on the FM that caused an anomaly. With the identified evolution operations, we are able to narrow down the search for fixes for the respective anomaly. As a consequence, we reduce explanation length by reducing it to the performed evolution operations.

Existing approaches to compute anomaly explanations identify constraints that cannot be satisfied and return them

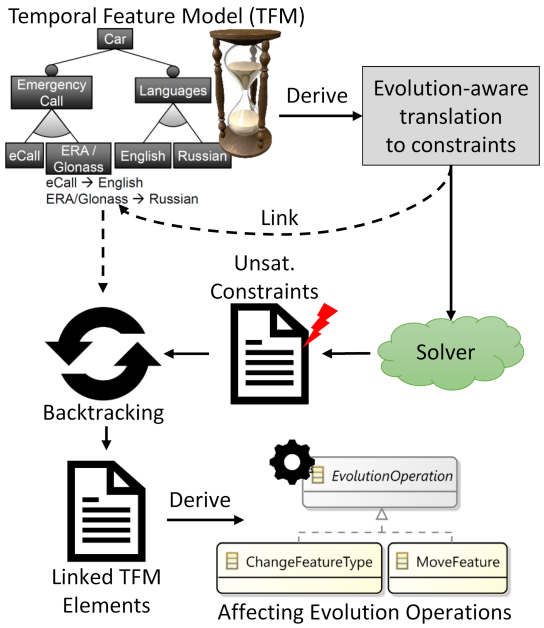


Figure 3. Evolution-Aware Explanation Workflow

as explanation. Evolution operations that caused an anomaly contribute to the existence of the constraints that cannot be satisfied. Thus, to identify the anomaly-causing evolution operations, we need to identify the evolution operations that have contributed to the constraints of the explanation. Figure 3 shows the general workflow of the evolution-aware anomaly explanation. To compute an explanation for an anomaly in the evolution history and to identify evolution operations that caused them, we reuse results from our evolution-aware anomaly detection (cf. Section 3) which identifies the date of anomaly introduction. For the date of anomaly introduction, we derive all temporally valid constraints from the TFM. At the same time, we link the elements from which the constraints have been computed. For instance, in the running example, if the constraint for the child feature of feature `Car` are computed for the evolution step t_2, the group and all of its sub features are linked to the translated constraint.

In the next step, we generate the explanation in terms of unsatisfiable constraints using the same formulas as before (cf. Section 3.1). After we generated an explanation, we need to identify the evolution operations that contributed to constraints of that explanation. To this end, we backtrack the TFM elements that we linked in the first step to the constraints. Then, to identify evolution operations that affected these elements, we reuse the information on evolution stored in the TFM. With TFMs, we have direct access to the elements that have been valid in the evolution step before the one for which we compute the constraints. Thus, if an element changed during evolution, we can derive the respective evolution operation. In particular, we are able to derive

the following operations: adding and removing cross-tree constraints, adding and removing features, changing a feature type, moving a feature into another group, and changing a group type. As the anomaly has been introduced by performing the identified evolution operations at the considered evolution step, they can be used to narrow down a fix for the anomaly. Thus, we can reduce the explanation by only presenting the evolution operations as anomaly cause.

In the running example, this would look like the following. For the *false-optional* feature anomaly at t_2 of the feature `ERA/Glonass`, we would retrieve the following constraints as explanation from the solver:

```
1 Car
2 Car → EmergencyCall ∧ Languages
3 EmergencyCall → eCall ⊕ ERA/Glonass
4 Diagnostic ↔ ERA/Glonass
5 Car → Diagnostic
```

Listing 3. Constraints of the Explanation for the Dead Feature Anomaly of the Running Example

During the translation process for the SMT solver query, we would link the constraints to the respective TFM elements. For instance, the last constraint results from the feature type of feature `Diagnostic` as it is *mandatory* at t_2 and, thus, we link the constraint to the feature `Diagnostic`. For each constraint, we analyze the TFM elements and check whether their temporal validity starts at t_1. In this case, we would identify that the type of feature `Diagnostic` has changed from *optional* to *mandatory* which is also the cause of this anomaly. Thus, we would reduce explanation complexity from five constraints to one evolution operation.

5 Evaluation

We evaluate the anomaly analyses for past and future FM evolution by four means. First, we show its feasibility by providing tool support for the evolution-aware anomaly detection, the evolution operation derivation, and the inspection of anomalies and explanations. Second, we qualitatively evaluate our method by verifying whether we correctly identify all evolution operations leading to an anomaly. Third, we quantitatively evaluate whether our method is applicable for real-world large-scale FMs and their evolution and verify whether we can improve performance due to reuse enabled by our evolution-aware encoding. Fourth, we evaluate to which extent we are able to reduce anomaly explanation complexity by using the case studies from the qualitative and quantitative evaluation and a third real-world case study.

5.1 Implementation

In this section, we show feasibility of our method by providing an implementation by means of two open-source tools. The first tool implements the solving part described in Section 3.2 and retrieval of explanation for unsatisfiable queries. The second tool allows to translate the entire TFM evolution into a satisfiability problem for a solver, following the method we described in Section 3.1. Moreover, we implemented the linking to TFM elements when translating it to constraints to retrieve anomaly explanations and the retrieval of evolution operations from unsatisfiable constraints as we described in Section 4. Finally, this tool allows anomaly and explanation inspection.

As we described in Section 3.2, standard SAT solvers supporting only propositional formulas are not well suited to analyze the satisfiability requests with the encoded evolution history that requires an Integer variable. Thus, we decide to use an SMT solver to be able to reason on the evolution variable. In particular we chose the first tool, an open-source and publicly available tool[2], extending it to support the previously mentioned analysis for detecting the features anomalies. The first tool is implemented as webservice and uses the Z3 SMT solver as backend.

The second tool is an extension of our existing tool suite and is open-source and publicly available[3] as well. It uses the first tool as solving engine and translates TFMs in the requested input format following the specifications of Section 3.1. After detecting anomalies using the first tool, the second tool shows the detected anomalies in an overview. Figure 4 shows the TFM of the running example and the

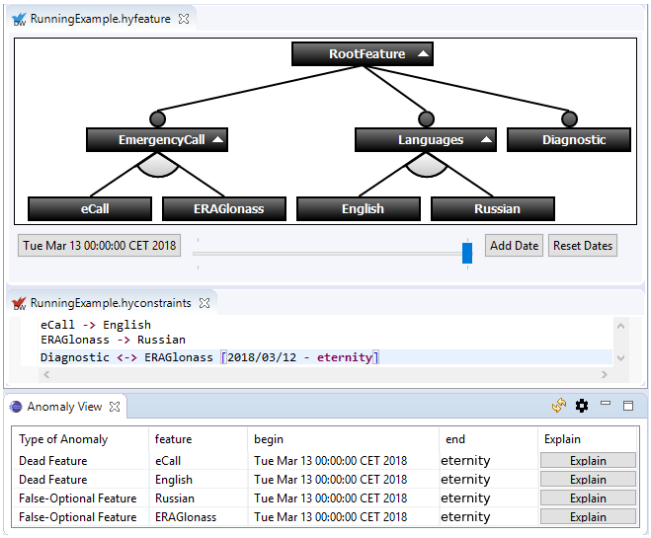


Figure 4. Screenshot of Detected Anomalies in our tool

view of the detected anomalies at the bottom. For each anomaly, the type, the affected feature, the duration of the anomaly is displayed.

Additionally, for each anomaly a button to start its explanation is visible. If this button is used, the translation of the TFM is started again and the linking of the translated

[2]Blinded for submission
[3]Blinded for submission

to TFM elements (cf. Section 4) is performed. Then, the first tool is contacted again and queried for an explanation. the first tool provides the set of unsatisfiable constraints for that respective anomaly. Since an anomaly may have multiple explanations, based on the internal search heuristics used for the SMT solver, we provide just one of them because at least one of the constraints in the returned set must be changed or removed to fix the anomaly. Using a backtracking mechanism, relevant TFM elements for that explanation are identified and evolution operations affecting these elements are derived. Afterwards, the explanation and the identified evolution operations are presented.

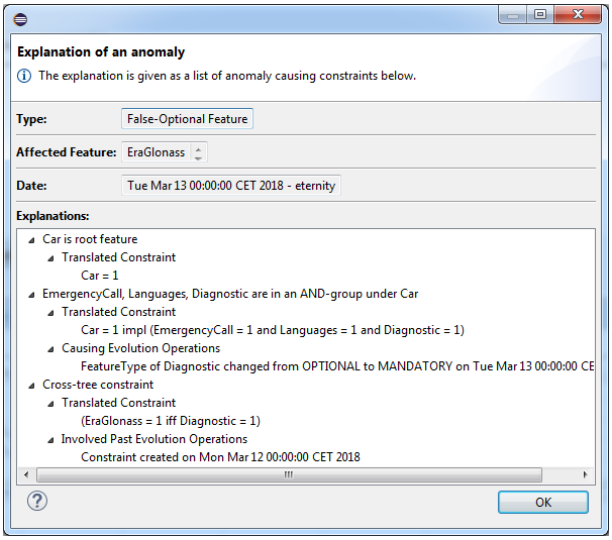


Figure 5. Anomaly Explanation from the Running Example

Figure 5 shows the explanation and evolution operation for the running example. Currently, all unsatisfied constraints are still listed for evaluation purposes. The most relevant part is the identified evolution operation under "Causing Evolution Operations" as this is an operation that lead to the analyzed anomaly. In a productive environment, all constraints except the causing evolution operations could be hidden in the first place.

5.2 Qualitative Evaluation

In the qualitative evaluation, we investigate whether we are able to identify and explain anomalies and pose the following research questions:

RQ-A-1: Do we identify all anomalies in FM evolution histories?

RQ-A-2: Do we correctly identify evolution operations performed during FM evolution that lead to anomalies?

As a case study to answer these research questions, we use the real-world SPL *Body-Comfort System (BCS)*, originally presented by Oster et al. [34] and extended with evolution by Nahrendorf et al. [28]. The original version of the BCS

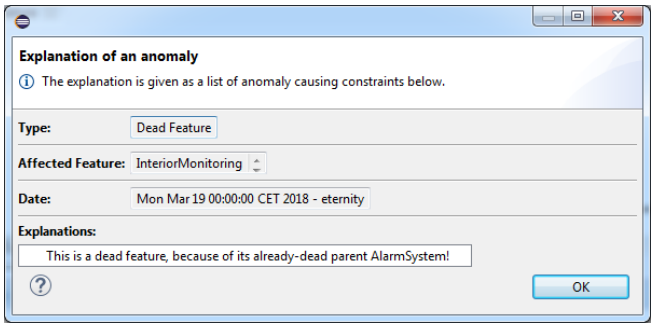


Figure 6. Relation Between two Anomalies of the Qualitative Evaluation

contained 28 features [34]. Nahrendorf et al. introduced four evolution steps resulting in 49 features after the last evolution step [28]. This case study is suitable for our evaluation as it does not have any anomalies, it already has multiple evolution steps and the FM is small enough to understand its evolution and . The first point is important as we need a ground truth for anomalies and the evolution operations performed leading to anomalies. For this purpose, we manually perform evolution operations introducing anomalies. The fact that the FM already has an evolution history is important so that we can verify whether we can correctly find all anomalies and their date of introduction and to verify that we do not have incorrect other already performed evolution operations in the explanations.

We create 12 different evolution scenarios of the BCS FM. In each of these scenarios, we manually introduce one anomaly. As some anomalies may entail other anomalies, we consider these additional anomalies as well. For instance, if a feature of an *alternative* group becomes false-optional, all of its sibling features become dead as a consequence. In particular, we create 6 dead feature anomalies and 6 false-optional feature anomalies. The explicitly introduced anomalies caused 11 further anomalies. For each of the evolution scenarios, we document which evolution operations we performed in the TFM editor in the description of the evolution scenario. All the data related to the evolution operation description, the corresponding requests for the solving engine and the results can be found in our online repository.[4]

To answer **RQ-A-1**, we investigate whether all of our seeded anomalies and also additionally entailed anomalies are found and the correct date of anomaly introduction is provided. For our case study, we are able to identify all anomalies, to classify them correctly and to provide the correct date of anomaly introduction. The results of this evaluation can be investigated using our online repository (cf. footnote 4).

Regarding **RQ-A-2**, we need to verify whether the identified evolution operations in the explanation match those

[4] https://gitlab.com/evolutionexplanation/evolutionexplanation

provided in the description for each scenario for each anomaly. Moreover, the evolution operations of the explanations for the anomalies additionally entailed by the seeded anomalies should be the same as for the seeded anomalies themselves. In the considered evolution scenarios, all identified evolution operations in the anomaly explanations matched the evolution operations actually performed in the editor. Other irrelevant evolution operations for the anomalies (e.g., those performed at a different evolution step) are not listed. For the anomalies entailed by the originally seeded anomalies, the explanations were the same except for some minor differences resulting from the FM structure.

Additionally, we are able to relate anomalies that imply other anomalies. For instance, if a certain feature is dead, all of its child features are dead as well. We detect this relation and make developers aware of it. Figure 6 shows how we present such a relation using two anomalies of one of our analyzed evolution scenarios. This explanation states that the feature `InteriorMonitoring` is dead because its parent `AlarmSystem` is dead, thus resulting in all its children being dead as well. The results indicate that we are able to identify all dead and false-optional feature anomalies in arbitrary FM evolution scenarios and that we provide the correct evolution operations that lead to those anomalies.

5.3 Quantitative Evaluation

In the quantitative evaluation, we investigate whether our method is applicable for large-scale real-world FM evolution. As we are able to reuse the solver thanks to our evolution-aware encoding, we are see potential for an improved performance. Thus, we analyze whether the incremental reuse of constraints for the analyses increases the performance. To this end, we pose the following research questions:

RQ-B-1: Is our method applicable for large-scale real-world FM evolution?

RQ-B-2: Does the evolution-aware analysis (i.e., reuse of constraints) for FM histories increase the performance?

As a case study to answer these research questions, we use the real-world FMs *Automotive02* and *FinancialServices1* with real-world evolution from the FeatureIDE example repository.[5] The evolution history of Automotive02 contains 4 versions. The Automotive02 FM contains between 14,010 (version 1) and 18,616 (version 4) features and between 666 (version 1) and 1,369 (version 4) cross-tree constraints. The evolution history of FinancialServices1 contains 10 versions. The FinancialServices1 FM contains between 557 (version 1) and 774 (version 10) features and between 1,001 (version 1) and 1,148 (version 9) cross-tree constraints. We evaluated each evolution step on its own as well as the evolution-aware analyses of the TFM. To analyze the merged evolution history, we imported all single evolution steps and integrated them into one TFM.

We deployed our solving engine as Docker container on virtual machines provided by an OpenStack private cloud[6]. Each virtual machine used Ubuntu 17.10, had 4 virtual cores and 8/16 GB RAM. We repeated each experiment 5 times and used the average values to minimize random factors.

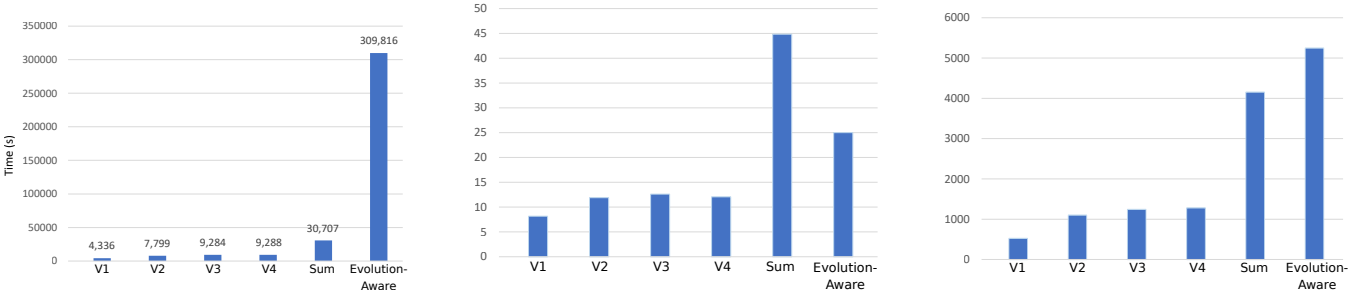
In the following, we only consider computation times of the solving engine backend. As the solving engine is running as a cloud service, it has to parse the requests beforehand which took for Autmotive02 between ~ 30 seconds (average for single evolution steps) and ~ 79 seconds (average merged evolution history) and for FinancialServices1 between ~ 6 seconds (average for single evolution steps) and ~ 23 seconds (average merged evolution history). The results for V1 – V10 show the computation times for analyzing each of these evolution steps on their own. The next to the last result shows the aggregated computation time for analyzing all evolution steps. The last result of each diagram shows the computation time for the evolution-aware analysis of the entire FM history. All data and results can be found in our online repository (cf. footnote 4).

In the first version of our solving engine and our experiments, we used an Integer encoding of the feature variables (i.e., feature instead or being considered Boolean values where considered to be integers in the $\{0, 1\}$ domain). Figure 7a shows the results of detecting feature anomalies using the Integer encoding. As can be seen, the computation times were extremely high. For the single evolution steps, finding all feature anomalies took in average more than 2 hours. For the merged model, it even took more than 86 hours. This might be the price to pay when analyzing cardinality-based FMs [7] where the Integer encoding may become necessary. As in our experiments, we are only dealing with FMs without cardinalities, we integrated the possibility to use a Boolean encoding for the feature variables to achieve acceptable computation times.

Figure 7b shows the performance of detecting void FM anomalies for the Automotive02 case study and using Boolean encoding. The average computation time for the void FM analyses for each evolution step is ~ 11 seconds. As can be seen, the sum of analyzing all individual evolution steps (~ 45 seconds) significantly exceeds the evolution-aware analysis (~ 25 seconds). Figure 7c shows the results for the feature-anomaly analyses for the Automotive02 case study using Boolean encoding. In this case, the average computation time for each evolution step is ~ $1,038$ seconds. For this analysis, the sum analyzing all individual evolution steps (~ $4,153$ seconds) is less than the evolution-aware analysis (~ $5,245$ seconds).

[5] https://github.com/FeatureIDE/FeatureIDE/tree/release3.5/

[6] Blinded for submission.

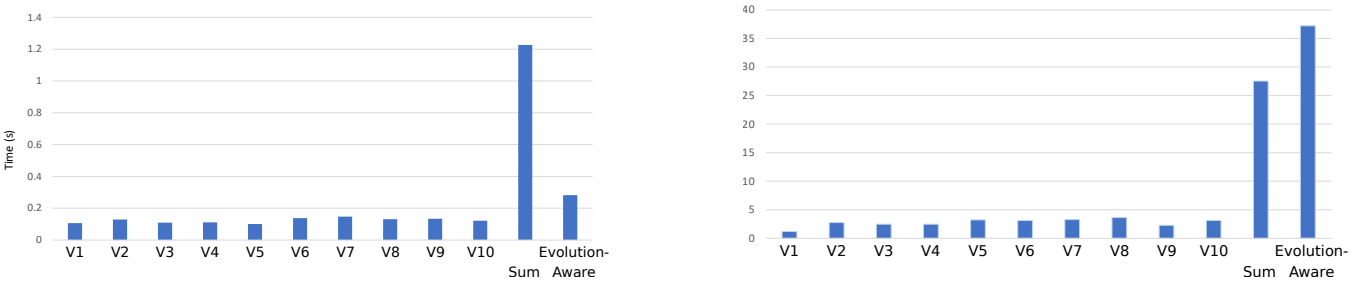


(a) Void Feature Model Detection using Integer Encoding

(b) Void Feature Model Detection using Boolean Encoding

(c) Feature Anomaly Detection using Boolean Encoding

Figure 7. Anomaly Detection Computation Times for Automotive02



(a) Void Feature Model Detection

(b) Feature Anomaly Detection

Figure 8. Anomaly Detection Computation Times for FinancialServices1

Except for the analyses using Integer encoding, we performed the same performance measurements for the FinancialServices1 case study. Figures 8a, 8b show the results of the void FM and feature-anomaly detection computation times for the FinancialServices1 case study. The average computation times for each individual evolution steps are ~ 0.12 seconds (void FM analysis) and ~ 2.8 seconds (feature-anomaly analyses). Similar to the Automotive02 case study, the sum of the individual computation times is significantly higher for the void FM analyses compared to the evolution-aware analyses but for the feature-anomaly detection, the evolution-aware analyses is slower.

To answer **RQ-B-1**, we can conclude that our method is applicable for large-scale real-world FM evolution. Even if we have computation times around 5, 245 seconds (cf. Figure 7c), it is an acceptable effort for analyzing the entire evolution history of such a large model. Compared to FeatureIDE, this still takes more time. In FeatureIDE, checking each evolution step of the Automotive02 for voidness takes a summed up computation time of ~ 0.53 seconds and searching for feature feature anomalies in each evolution step takes a summed up computation time of ~ 293 seconds. We performed the experiments with FeatureIDE on a Windows 10 machine with an Intel Core i7-5600U @ 2.60GHz and 12GB of RAM. We believe that this is due to the fact that FeatureIDE exploits the tree structure of the FM to perform many optimizations that unfortunately are not possible with our solving engine that relies on a more general declarative specification of the FM. More importantly, FeatureIDE uses a different notion of false-optional features than the one proposed by Benavides et al. [3]. In particular, in FeatureIDE, a feature is false-optional if it must be selected if its parent is selected despite having the type *optional*. Despite this difference in performance and definition of anomalies, compared to FeatureIDE, we provide additional information such as the introduction date of an anomaly and the causing evolution operations. Thus, with our method, we significantly increase the support for developers in fixing anomalies in the evolution history.

Regarding **RQ-B-2**, we do not have an unambiguous answer. As Figures 7b, 7c, 8a, 8b show, performing the evolution-aware analysis (i.e., for the merged model) is sometimes faster than performing the sum of all single step analyses. For the feature-anomaly analysis, the cumulative times taken by the single step analyses are faster. Unfortunately, since we are trying to solve NP-hard problems, we are not yet able to predict under which circumstances the evolution-aware analyses is faster. However, for the cases for which the evolution-aware analysis is slower, the factor is not that high, but for the cases for which the evolution-aware analysis is faster, the difference is significant.

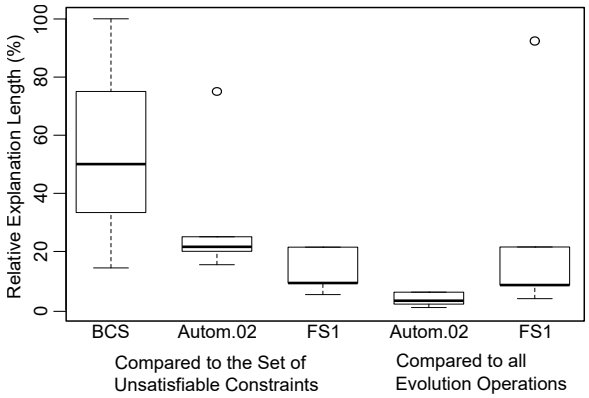


Figure 9. Anomaly explanation complexity reduction rates.

5.4 Explanation Reduction Evaluation

Explanations from existing methods can grow very large and during evolution, even single evolution operations can cause anomalies. Thus, understanding these explanations is a complex task but if the anomaly has been introduced during evolution, we argue that the evolution operations causing the anomaly are easier to understand for developers as they are potentially fewer but more expressive. In this part of the evaluation, we evaluate whether our method is able to reduce anomaly explanation complexity. To this end, we pose the following research question:

RQ-C-1: To which percentage are we able to reduce anomaly explanation complexity?

As case studies, we use the FMs and their evolution histories of the qualitative (Body-Comfort System (BCS)) and quantitative evaluation (Automotive02 and FinancialServices1). In particular, we analyze the anomalies that have been introduced during evolution as for anomalies that have been there from the beginning, no evolution operations exist that have introduced these anomalies and, thus, no explanation complexity reduction is possible. For the BCS, we analyze 17 anomalies as we also consider anomalies that arised due to other anomalies (cf. Section 5.2). Most anomalies of the Automotive02 case study have already been there from the beginning and, thus, we cannot use them for this evaluation. As a consequence, we analyzed the explanations of 6 anomalies. During the evolution of the FinancialServices1 FM, 9 anomalies were introduced. As a baseline, we use the number of unsatisfiable constraints in explanations that other methods would provide. Between 3 – 7 constraints are unsatisfiable for the anomalies of the BCS, between 4 – 13 constraints are unsatisfiable for Automotive02 and between 11 – 98 constraints are unsatisfiable for FinancialServices1. The fact that the maximum number of unsatisfiable constraints is larger for FinancialServices1 than for Automotive02 shows that even smaller FMs may have large explanations.

To analyze to which percentage we are able to reduce explanation length, we compare the number of unsatisfiable constraints with the number of identified evolution operations causing the respective anomalies. Another approach for identifying the cause for anomalies could be to investigate the difference and, thus, the performed evolution operations between two FM versions [6]. However, this method would provide the set of all evolution operations between two FM versions. To compare our method with methods reasoning about FM differences, we measure the percentage of identified evolution operations causing an anomaly compared to the number of all evolution operations performed at the introduction date.

Figure 9 shows the relative explanation length of our method, whereas lower numbers are better. The first three plots compare the number of identified causing evolution operations with the set of unsatisfiable constraints. The last two plots compare the number of identified causing evolution operations with the number of all evolution operations performed at the date of anomaly introduction. For the latter comparison, we did not include the BCS as we explicitly performed single evolution operations leading to anomalies (cf. subsection 5.2) and, thus, it would always be 0% by design.

As can be seen, the relative explanation length compared with the number of unsatisfiable constraints for the BCS are between $\sim$ 14% – 100%, for Automotive02 between $\sim$ 15% – $\sim$ 75% and for FinancialServices1 between $\sim$ 5% – $\sim$ 21%. The longest explanation contained 98 unsatisfiable constraints for FinancialServices1 and it we identified 5 causing evolution operations. In 2 cases of the BCS, the number of identified evolution operations is equal to the number of constraints in the original explanation and, thus, no reduction is achieved. However, in 9 cases of the BCS and in 3 cases of the Automotive02 case study, we are able to reduce complexity by more than half.

For the comparison between causing evolution operations with all evolution operations, we achieve even more significant reduction rates. For Automotive02, the relative explanation length is between $\sim$ 0.6% – $\sim$ 6% and for FinancialServices1 it is between $\sim$ 4% – $\sim$ 92%. The reason for the low relative explanation length for Automotive02 are most likely as between the FM versions many operations were performed. The most significant reduction was achieved for the evolution step between version 1 and version 2 for which 169 evolution operations were performed and we identified 1 operation as cause for a dead feature anomaly. In contrast, one anomaly in FinancialServices1 was caused by almost all performed evolution operations (12 out of 13). To answer **RQ-C-1**, we claim that the length of most of the explanations of the anomalies are significantly reduced by using our method.

5.5 Threats to Validity

The results of the qualitative and quantitative evaluation are subject to threats to validity. In the qualitative evaluation,

we manually seeded anomalies in the evolution history of an FM and verified whether all anomalies were found and the evolution operations of the explanations matched the actually performed evolution operations. We used this setup as we only had access to SPLs with real-world evolution but without a meaningful number of anomalies that was still analyzable manually to find all existing anomalies. The internal validity might be biased as we probably did not know all introduced anomalies and evolution operations performed. To mitigate this threat, we used an FM with a moderate size so that a manual analysis was feasible and we documented all evolution operations we performed in the editor which we used as ground truth. Moreover, in the quantitative evaluation, we verified that we where able to detect the same anomalies using FeatureIDE manually for each evolution step[7].

In the explanation reduction evaluation, we measure the rate of explanation length reduction. In this case, we compare the number of constraints of the original explanation with the number of identified evolution operations. Thus, we assume that understanding a constraint of the original explanation is as complex as understanding an evolution operation which is an internal threat to validity. However, our experiences with explanations has shown that understanding evolution operations is even easier for developers than understanding constraints as the operations are common to developers.

The anomalies and evolution operations analyzed in the qualitative evaluation may not be representative for other evolution scenarios or SPLs. To mitigate this threat, we analyzed 12 different anomalies and evolution operations leading to these anomalies for a real-world FM with evolution.

The results of the quantitative evaluation may be subject to performance volatility resulting in falsified results. To mitigate this threat, we performed each of these analyses multiple times and used the average values. The only analyses we only performed once were those for which the results were clear, i.e., the experiments using the Integer encoding in Figure 7a.

The quantitative evaluation may not be representative as we analyzed the evolution history of only two FMs. We only used two FMs as we did not have access to other large-scale FMs with their evolution history. To mitigate this threat, we explicitly used two real-world FMs that are publicly available.

The explanation reduction evaluation may not be representative as we only analyzed 32 anomalies in total. To mitigate this threat, we used anomalies of the evolution of two real-world FM and their real-world evolution. Moreover, when discussing the results, we distinguished between the case studies to see which results stem from seeded anomalies and which stem from real-world anomalies. As the results of the real-world case studies are similar and even better than the one from the qualitative evaluation, we are encouraged that our results are representative. Another threat is that developers might perform analyses more often than just between the considered FM versions which would result in fewer overall evolution operations. To mitigate this threat, we explicitly used the FinancialServices1 case study which versions are retrieved on a monthly base and, thus, are relatively fine-grained.

[7] In particular, we detect the same dead features but more false optional due to the different definition of false-optional features in FeatureIDE.

6 Related Work

Many approaches exist to analyze FMs with a high degree of optimization [2–4, 19, 27, 47]. Multiple approaches are able to detect anomalies but are not able to explain them [14, 51]. Other approaches are able to provide explanations for anomalies [2, 10, 17, 18, 20, 49, 50]. Some of these approaches focus on contradictions in the configuration process [2, 18]. Lesta et al. [20] are able to detect and explain dead features and false-optional features in attributed feature models. Trinidad et al. [49, 50] provide the tool suite FAMA which detects and explains dead and false-optional features as well. However, explanations are abbreviated which even increases the problem of understanding the cause of the anomalies. The method of Felferning et al. [10] to explain anomalies does not relate explanations to the FM structure, as we do. Finally, Kowal et al. [17] provide a method to detect and explain dead and false-optional features. Additionally, they also detect redundant constraints and highlight which parts of the explanations are more important than others. They also provide tool implementation in FeatureIDE. However, none of the previously mentioned methods incorporates evolution. As a consequence, they are not able to analyze the entire evolution history of an FM or its future planning, to pinpoint the introduction date of anomalies, and they are not able to determine evolution operations which caused anomalies.

Multiple techniques were published dealing with the detection of range inconsistencies in cardinality-based FMs [39, 52]. As the FMs we consider are special cases of cardinality-based FMs (i.e., each feature has a cardinality of 1), some of the analyses are applicable to the FMs we consider and should also be capable of detecting dead and false-optional features. Quinton et al. also define which evolution operations can lead to range inconsistencies and explain the found range inconsistencies [39]. However, both approaches do not analyze the evolution history of an FM and do not incorporate evolution operations in their explanations.

To capture evolution of product lines, Schubanz et al. [44, 45], Pleuss et al. [37] and Botterweck et al. [5] introduce EvoFM and the EvoPL framework. With their method it is possible to model and plan FM evolution. They also provide techniques to check model and configuration consistency.

However, they do not incorporate evolution in their analyses and they do not detect FM anomalies.

Alves et al. present a theory for FM refactorings and a set of refactoring operations [1]. Similarly, Neves et al. propose a theory and a catalogue for safe evolution templates [29, 30]. In contrast to Alves et al. [1], they also incorporate consistent co-evolution of FMs, mapping and artifacts. However, the refactorings and the safe evolution templates may not result in FM anomalies. Thus, when these operations are applied, it would not be necessary to analyze the FM. Sampaio et al. [42] relaxed the notion of safe evolution templates of Neves et al. [29] and present the concept of partially safe evolution templates. However, as these evolution templates are only safe for a subset of configurations, they may introduce FM anomalies. Similarly, Seidl et al. present a method and templates for co-evolving FMs and their mapping to implementation artifacts [46]. Neither Sampaio et al. [42] nor Seidl at al. [46] analyze FMs to find anomalies.

Guo et al. provide an approach to efficiently analyze the consistency of an FM in the presence of evolution [11, 12]. They do not analyze the entire FM and its history again, but only focus on parts changed by evolution operations since the last check. However, this requires that the FM is valid before checking it again. Moreover, they do not find dead or false-optional features and do not provide any explanations for inconsistencies. Nevertheless, it might be sensible to investigate whether their method can be combined with ours.

Several techniques exist to reason about FM differences [6, 8, 48]. These differences can be assumed as changes between two evolution steps. To this end, Dintzner et al. provide a set of operations they identified in the Linux kernel variability model which they are capable to detect with their tool FMDiff [8]. However, this approach does only work for KConfig variability models. None of these techniques is capable of detecting FM anomalies. Nevertheless, the approaches may provide additional information about the evolution which can be used to analyze the FM history more efficiently. Tartler et al. analyzed the variability model of the Linux kernel and searched for anomalies [40]. The work has proven to be applicable to large-scale models. However, it is again specific for the Linux kernel variability model and does not incorporate evolution or provide anomaly explanations.

Lity et al. use the concept of higher-order deltas to capture the evolution of implementation artifacts in one 175% model [22, 23] – similar to TFMs. Evolution of the implementation artifacts may also have impact on FMs. Thus, in future work, analyses might consider co-evolution of implementation artifacts and FMs.

Guthmann et al. [13] and Liffiton et al. [21] provide methods to retrieve minimal explanations. For this purpose, Guthmann et al. [13] compute the minimal unsatisfiable core using an SMT solver and Liffiton et al. [21] provide algorithms to compute minimal unsatisfiable subsets of constraints. In our method, we use similar methods to compute the unsatisfiable constraints.

7 Conclusion

We presented a method to analyze past and future evolution histories of FMs encoded in Temporal Feature Models (TFMs). We proposed a method to encode the entire TFM evolution history into one request for a solver by introducing evolution as a distinct variable. Using such requests, we identified FM anomalies, pinpointed their date of introduction and identified the anomaly-causing evolution operations. Additionally, we provide tools implementing this method, allowing easy inspection of anomalies and their explanations. We performed three evaluations: First, we qualitatively analyzed whether we were able to detect all anomalies in the evolution history of a real-world FM and provide the respective correct explanations. Second, we quantitatively analyzed whether our method is applicable for real-world evolution of a large-scale FM and of a medium-sized FM. To this end, we measured the performance of our method for each individual evolution step and compared it to the evolution-aware analysis. Third, we measured how using the identified evolution operations as anomaly explanations reduced explanation length. The results of our evaluations indicate that we are able to detect all anomalies in FM evolution histories and provide the correct explanations for it in a reasonable amount of time. Additionally, we are able to reduce explanation length significantly.

This work raises several further research opportunities. To investigate the increase of comprehensibility and the support for fixing anomalies in the evolution history, we want to perform a supervised experiment with two user groups: one using explanations without information about the evolution history and another group using the additional information provided by our method. To support the anomaly detection for context-adaptive SPLs in presence of evolution, we plan to combine the results of this work with context-aware analyses. Additionally, we want to integrate the detection of more relations between anomalies (e.g., features that became dead because another feature became false-optional) or anomalies related to attributes (e.g., an attribute value that may never be selected). Finally, we want to perform more case studies to understand in which cases the evolution-aware analysis is faster.

References

[1] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. 2006. Refactoring Product Lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*. ACM, New York, NY, USA, 201–210. https://doi.org/10.1145/1173706.1173737

[2] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*. Springer-Verlag, Berlin, Heidelberg, 7–20. https://doi.org/10.1007/11554844_3

[3] 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.

[4] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)*. Springer-Verlag, Berlin, Heidelberg, 491–503. https://doi.org/10.1007/11431855_34

[5] 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 '10)*. ACM, New York, NY, USA, 24–31. https://doi.org/10.1145/1808937.1808941

[6] 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 Engg.* 23, 4 (Dec. 2016), 687–733. https://doi.org/10.1007/s10515-015-0185-3

[7] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice* 10 (2005), 7–29.

[8] Nicolas Dintzner, Arie Deursen, and Martin Pinzger. 2017. Analysing the Linux Kernel Feature Model Changes Using FMDiff. *Softw. Syst. Model.* 16, 1 (Feb. 2017), 55–76. https://doi.org/10.1007/s10270-015-0472-2

[9] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. 2009. Using First Order Logic to Validate Feature Model. In *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings*. 169–172.

[10] Alexander Felfernig, David Benavides, José A. Galindo, and Florian Reinfrank. 2013. Towards Anomaly Explanation in Feature Models. In *Proceedings of the 15th International Configuration Workshop, Vienna, Austria, August 29-30, 2013*. 117–124.

[11] Jianmei Guo and Yinglin Wang. 2010. Towards Consistent Evolution of Feature Models. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 451–455.

[12] Jianmei Guo, Yinglin Wang, Pablo Trinidad, and David Benavides. 2012. Consistency maintenance for evolving feature models. *Expert Systems with Applications* 39, 5 (2012), 4987–4998.

[13] Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. 2016. Minimal unsatisfiable core extraction for SMT. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. 57–64.

[14] Adithya Hemakumar. 2008. Finding Contradictions in Feature Models. In *Proceedings of the 12th International Software Product Line Conference (SPLC)*. 183–190.

[15] 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.

[16] 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. https://doi.org/10.1145/3106237.3106252

[17] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining anomalies in feature models. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*. 132–143. https://doi.org/10.1145/2993236.2993248

[18] Dean Kramer, Christian Severin Sauer, and Thomas Roth-Berghofer. 2013. Towards Explanation Generation using Feature Models in Software Product Lines. In *Proceedings of 9th Workshop on Knowledge Engineering and Software Engineering (KESE9) co-located with the 36th German Conference on Artificial Intelligence (KI2013), Koblenz, Germany, September 17, 2013*.

[19] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA. https://doi.org/10.1145/3180155.3180159

[20] Uwe Lesta, Ina Schaefer, and Tim Winkelmann. 2015. Detecting and Explaining Conflicts in Attributed Feature Models. In *Proceedings 6th Workshop on Formal Methods and Analysis in SPL Engineering, FMSPLE@ETAPS 2015, London, UK, 11 April 2015*. 31–43.

[21] Mark H. Liffiton and Karem A. Sakallah. 2008. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *J. Autom. Reasoning* 40, 1 (2008), 1–33.

[22] Sascha Lity, Matthias Kowal, and Ina Schaefer. 2016. Higher-order Delta Modeling for Software Product Line Evolution. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD 2016)*. ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/3001867.3001872

[23] Sascha Lity, Sophia Nahrendorf, Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2018. 175Artifacts. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2018)*. ACM, New York, NY, USA, 27–34. https://doi.org/10.1145/3168365.3168369

[24] 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. http://dl.acm.org/citation.cfm?id=1885639.1885653

[25] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2017. Anomaly Detection and Explanation in Context-Aware Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25-29, 2017*. 18–21. https://doi.org/10.1145/3109729.3109752

[26] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[27] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Software Product Line Conference (SPLC '09)*. Carnegie Mellon University, Pittsburgh, PA, USA, 231–240. http://dl.acm.org/citation.cfm?id=1753235.1753267

[28] Sophia Nahrendorf, Sascha Lity, and Ina Schaefer. 2018. *Applying Higher-Order Delta Modeling for the Evolution of Delta-Oriented Software Product Lines*. Technical Report. TU Braunschweig. https://www.isf.cs.tu-bs.de/cms/team/lity/TUBS_Report_2018-01_Nahrendorf_et_al.pdf

[29] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demóstenes Sena, and Uirá Kulesza. 2015. Safe evolution templates for software product lines. *Journal of Systems and Software* 106 (2015), 42–58. https://doi.org/10.1016/j.jss.2015.04.024

[30] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, 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 '11)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/2047862.2047869

[31] 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. https://doi.org/10.1145/3023956.3023962

[32] 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. https://doi.org/10.1145/2866614.2866625

[33] 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 B (SPLC '18)*. ACM, New York, NY, USA.

[34] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. 2011. Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11)*. ACM, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/2019136.2019143

[35] 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. https://doi.org/10.1145/2430502.2430526

[36] 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. https://doi.org/10.1145/2377816.2377825

[37] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. 2012. Model-driven Support for Product Line Evolution on Feature Level. *J. Syst. Softw.* 85, 10 (Oct. 2012), 2261–2274. https://doi.org/10.1016/j.jss.2011.08.008

[38] K. Pohl, G. Böckle, and F.J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer Berlin Heidelberg.

[39] Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien, and Goetz Botterweck. 2014. Consistency Checking for the Evolution of Cardinality-based Feature Models. In *Proceedings of the 18th International Software Product Line Conference - Volume 1 (SPLC '14)*. ACM, New York, NY, USA, 122–131. https://doi.org/10.1145/2648511.2648524

[40] Tartler Reinhard, Sincero Julio, Schröder-Preikschat Wolfgang, and Lohmann Daniel. 2009. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD '09)*. ACM, New York, NY, USA, 81–86. https://doi.org/10.1145/1629716.1629732

[41] LF Rincón, Gloria Lucia Giraldo, Raúl Mazo, and Camille Salinesi. 2014. An ontological rule-based approach for analyzing dead and false optional features in feature models. *Electronic notes in theoretical computer science* 302 (2014), 111–132.

[42] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2016. Partially Safe Evolution of Software Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*. ACM, New York, NY, USA, 124–133. https://doi.org/10.1145/2934466.2934482

[43] 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. https://doi.org/10.1007/s10009-012-0253-y

[44] Mathias Schubanz, Andreas Pleuss, Goetz Botterweck, and Claus Lewerentz. 2012. Modeling Rationale over Time to Support Product Line Evolution Planning. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12)*. ACM, New York, NY, USA, 193–199. https://doi.org/10.1145/2110147.2110169

[45] Mathias Schubanz, Andreas Pleuss, Ligaj Pradhan, Goetz Botterweck, and Anil Kumar Thurimella. 2013. Model-driven Planning and Monitoring of Long-term Software Product Line Evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*. ACM, New York, NY, USA, Article 18, 5 pages. https://doi.org/10.1145/2430502.2430527

[46] 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 - Volume 1 (SPLC '12)*. ACM, New York, NY, USA, 76–85. https://doi.org/10.1145/2362536.2362550

[47] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages. https://doi.org/10.1145/2580950

[48] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning About Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 254–264. https://doi.org/10.1109/ICSE.2009.5070526

[49] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Miguel Toro. 2006. Explanations for agile feature models. In *Procceedings of the 1st International Workshop on Agile Product Line Engineering (APLE'06)*. 44.

[50] Pablo Trinidad Martin-Arroyo. 2012. *Automating the analysis of stateful feature models.* Ph.D. Dissertation. University of Seville.

[51] Thomas von der Maßen and Horst Lichter. 2004. Deficiencies in feature models. In *Workshop on Software Variability Management for Product Derivation-Towards Tool Support*, Vol. 44.

[52] Markus Weckesser, Malte Lochau, Thomas Schnabel, Björn Richerzhagen, and Andy Schürr. 2016. Mind the Gap! Automated Anomaly Detection for Potentially Unbounded Cardinality-Based Feature Models. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering - Volume 9633*. Springer-Verlag New York, Inc., New York, NY, USA, 158–175. https://doi.org/10.1007/978-3-662-49665-7_10