Consistency-Preserving Evolution Planning on Feature Models

Adrian Hoff
a.hoff@tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Michael Nieke
m.nieke@tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Christoph Seidl
chse@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Eirik Halvard Sæther
eirikhsa@ifi.uio.no
University of Oslo
Oslo, Norway

Ida Sandberg Motzfeldt
idasmot@ifi.uio.no
University of Oslo
Oslo, Norway

Crystal Chang Din
crystald@ifi.uio.no
University of Oslo
Oslo, Norway

Ingrid Chieh Yu
ingridcy@ifi.uio.no
University of Oslo
Oslo, Norway

Ina Schaefer
i.schaefer@tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

ABSTRACT

A software product line (SPL) enables large-scale reuse in a family of related software systems through configurable features. SPLs represent a long-term investment so that their ongoing evolution becomes paramount and requires careful planning. While existing approaches enable to create an evolution plan for an SPL on feature-model (FM) level, they assume the plan to be rigid and do not support retroactive changes. In this paper, we present a method that enables to create and retroactively adapt an FM evolution plan while preventing undesired impacts on its structural and logical consistency. This method is founded in structural operational semantics and linear temporal logic. We implement our method using rewriting logic, integrate it within an FM tool suite and perform an evaluation using a collection of existing FM evolution scenarios.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software evolution**; *Software notations and tools.*

KEYWORDS

Software Product Lines, Software Evolution, Feature Models, Feature Model Evolution, Formal Semantics, Rewriting Logic, Structural Operational Semantics, Linear Temporal Logic

ACM Reference Format:
Adrian Hoff, Michael Nieke, Christoph Seidl, Eirik Halvard Sæther, Ida Sandberg Motzfeldt, Crystal Chang Din, Ingrid Chieh Yu, and Ina Schaefer. 2020. Consistency-Preserving Evolution Planning on Feature Models. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3382025.3414964

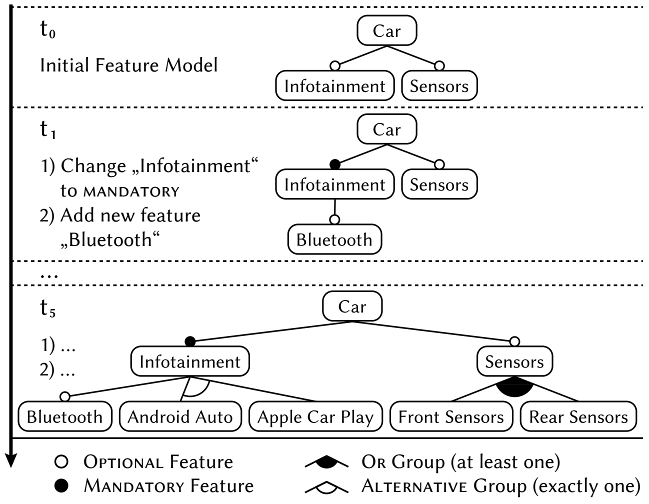


Figure 1: FM evolution plan for a configurable car system.

1 INTRODUCTION

A software product line (SPL) is a collection of similar software products that exploits the high similarity between individual products by organizing them into common and variable parts [46, 53]. A feature model (FM) organizes all user-visible characteristics, i.e., *features*, of an SPL in a hierarchical tree according to their interrelations [6]. An FM describes all possible configurations of an SPL, which is used not only to configure individual variants of the SPL but also for communication between (non-technical) stakeholders [48].

SPLs, like all software systems, have to undergo continuous evolution to remain relevant [12, 26, 29]. The development of complex and large-scale SPLs is a time-intensive and costly endeavor, which aims to benefit in the long term [8, 10, 11, 15]. Unforeseen complications throughout the evolution of an SPL may cause expensive deployment delays and poorly implemented last-minute fixes. Thus, SPL development should be supported by careful long-term evolution planning [10, 11].

Evolutionary changes to an SPL optimally begin with alterations to the FM [45, 48]. As FMs are also used for communication, they are development artifacts particularly suitable to use for planning [9]. We propose to exploit this by modeling the planned evolution of an SPL on FM level. For that purpose, we define a *feature model evolution plan* as a combination of (i) an initial FM version and (ii) an ordered sequence of intended edit operations, where each edit operation is scheduled for a concrete point of time in the future. Consequently, each planned edit operation serves as abstract, unimplemented, yet formally documented step of an overall FM evolution plan. While concepts and tools exist that allow to construct FM evolution plans, we identify two significant, yet unaddressed, barriers to a sensible and scaling usage of such plans:

Retroactively introducing a planned edit operation into an FM evolution plan as intermediate step can violate the *structural consistency* of a plan and, thus, make it unimplementable [45]. A retroactively introduced intermediate evolution step adds an operation that is scheduled to become effective before other already planned evolution operations. Consequently, a retroactively planned evolution step changes the basis of other already planned evolution steps. State-of-the-art concepts and analyses are not able to prevent inconsistencies arising from modifying planned evolution operations as they were not constructed for evolution planning but, at best, for tracking performed FM evolution. Respectively, state-of-the-art evolution-aware FM tools are only able to ensure the safe execution of an edit operation when it is appended to the end of an evolution time-line [41]. However, with a long-term planning horizon, short-notice changes to an FM may have to be integrated into the overall plan, e.g., due to unexpected obstacles in implementation or budgeting. Hence, adequate planning means must be able to accommodate for retroactive changes in any stage of a plan to have practical relevance. This renders state-of-the-art methods insufficient for FM evolution planning.

Similarly, performing changes to an FM evolution plan is hindered by the fact that operations are scheduled by their specific point in time alone, not taking into account the circumstances that lead to their scheduling, e.g., stemming from project management. For instance, existing FM planning concepts do not permit specifying logical dependencies of planned changes, e.g., that one change depends on another change so that they must not be scheduled in inverse order when adapting an evolution plan. We refer to this as *logical consistency* of an FM evolution plan. Due to these shortcomings, project-specific planning concerns are either documented only informally or are lost completely when solely placing an operation with a tentative point of time for realization in the evolution plan. This creates a risk for fatal planning errors when interdependencies between planned changes are disregarded as part of replanning.

In this work, we provide a concept for SPL evolution planning based on FMs that ensures the structural and logical consistency of an evolution plan even in the light of retroactively incorporated intermediate changes. In particular, we contribute an execution semantics based on structural operational semantics and linear temporal logic that is able to (i) consider each edit operation of an FM evolution for its consistent applicability and (ii) verify additional project-specific planning concerns. We provide an implementation of these concepts using rewriting logic and integrate them into an FM tool suite.

The rest of this paper is structured as follows: In Section 2, we present SPL techniques and formal methods that serve as foundation of our work. In Section 3, we motivate challenges in FM evolution planning and provide a formal definition of an FM evolution plan. In Section 4, we extend our example, analyze risks to the structural consistency of an FM evolution plan and provide a method that enables to prevent structural inconsistencies in an FM evolution plan. In Section 5, we build upon this basis to introduce a constraint language for defining project-specific concerns for an FM evolution plan to ensure its logical consistency. In Section 6, we evaluate our method with regard to its feasibility and performance on a collection of existing FM evolution scenarios. In Section 7, we elaborate on related work and, in Section 8, we conclude with an outlook to future work.

2 BACKGROUND

Our concept for consistency preserving evolution planning on FMs are part of SPL engineering and founded in formal methods. Hence, in this section, we first introduce SPL concepts and then provide essentials of the various forms of logic used in our solution.

2.1 Feature-Oriented Development

A **software product line** (SPL) manages a collection of similar software systems by reusing functionality shared between products in terms of *features* [46, 53]. Individual products can be derived from an SPL by assembling relevant realization artifacts according to a *configuration*, i.e., a valid selection of features.

A **feature model** (FM) manages configuration options of an SPL on conceptual level, i.e., without implementation details. For that purpose, an FM organizes all features of an SPL in a tree-like structure, where each feature can have an arbitrary number of child features. The relation between a feature and a collection of child features is referred to as *group*. A feature can only be added to a configuration if its parent feature is also part of the configuration. Additionally, most FM notations support assigning types to features and groups [16, 20]. A feature can be either OPTIONAL (i.e., *can* be selected if its parent feature is selected) or MANDATORY (i.e., *must* be selected if its parent feature is selected). A group can be assigned one of the types AND (i.e., select an *arbitrary* number of child features), OR (i.e., select *at least* one child feature), or ALTERNATIVE (i.e., select *exactly* one child feature). As basis for our later solution, we explicitly define a list of FM well-formedness rules that underly common feature modeling notations [16]:

WF1 A feature model has exactly one root feature.
WF2 The root feature must be mandatory.
WF3 Each feature has exactly one unique name, variation type and (potentially empty) collection of subgroups.
WF4 Features are organized in groups that have exactly one variation type.
WF5 Each feature, except for the root feature, must be part of exactly one group.
WF6 Each group must have exactly one parent feature.
WF7 Groups with types ALTERNATIVE or OR must not contain MANDATORY features.
WF8 Groups with types ALTERNATIVE or OR must contain at least two child features.

Table 1: Examples for syntax and semantics of LTL formulas

Operators	Explanation
$\bigcirc\,\phi$	ϕ has to hold at the next state
$\Diamond\,\phi$	ϕ eventually has to hold
$\Box\,\phi$	ϕ has to hold on the entire subsequent path
$\psi\ \mathrm{U}\ \phi$	ψ has to hold at least until ϕ becomes true, ϕ must hold at the current or a future point

A **feature diagram** is a graphical representation of an FM. Figure 1 shows multiple feature diagrams in the context of an FM evolution plan that we use as running example in Sections 3, 4 and 5.

2.2 Formal Semantics and Logic Frameworks

Structural operational semantics (SOS), commonly referred to as small-step semantics, is a form of formal semantics that describes how individual steps of a computation take place [2, 51]. An SOS specification is a set of inference rules, which can be defined in the following form:

$$\frac{Conditions}{State \Rightarrow State'}$$

It describes a transition from $State$ to $State'$ if all the $Conditions$ defined above the line are satisfied. We use SOS to define the semantics of (i) FM edit operations and (ii) an FM evolution plan.

Rewriting logic is a computational logic that can be used as a semantic and logical framework [1, 36]. A system built in rewriting logic consists of an equational logic that captures the system's state and a set of rewrite rules that realize its behavior. This way, SOS can be constructed in form of rewrite rules.

Linear temporal logic (LTL), also referred to as *linear-time temporal logic*, is a modal temporal logic that can be employed to express formulas over a path's future states [19, 31, 52]. LTL uses a model of natural numbers to represent time that are ordered sequentially and represent discrete states. In LTL, different expressions can be used that define properties of the path of states. For instance, it can be expressed that a condition is met in all existing states, will eventually be met in a future state, or remains met until another condition is met. Examples for syntax and semantics of LTL formulas can be found in Table 1, in which the formulas ψ and ϕ may also contain operators for negation, conjunction, disjunction and implication. We use LTL to verify whether project-specific concerns to the logical consistency of an FM evolution plan are met.

Maude is a programming language and logical framework that allows to specify and execute systems with the use of rewriting logic [13]. Furthermore, Maude offers a built-in model checker that allows to verify LTL formulas. We use Maude to provide an implementation of our formal concepts.

3 FEATURE MODEL EVOLUTION PLANS

Figure 2 highlights the challenges in consistently adapting the example FM evolution plan depicted in Figure 1. A car manufacturer plans to implement a variable car software system in form of an SPL. In its initial version, the car system is supposed to have optional, basic *Sensors* functionality (for parking assistance purposes) as well as an optional *Infotainment* system. The initial FM version covering

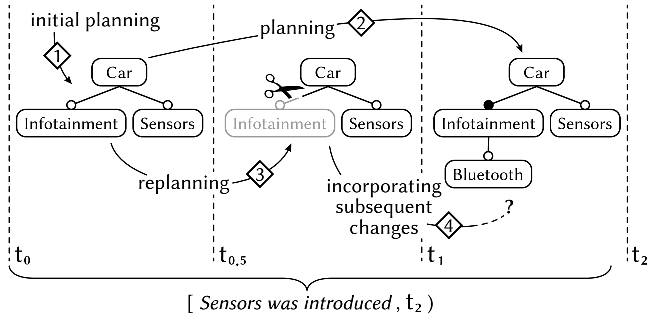


Figure 2: Constructing and extending an FM evolution plan. A retroactive intermediate change violates structural consistency.

these requirements is created for t_0. This process is depicted by arrow ⟨1⟩ in Figure 2, along with the resulting FM.

The car manufacturer observes that most customers configure cars with an *Infotainment* system. Furthermore, customers request configurable *Bluetooth* functionality. The car manufacturer thus plans to make their *Infotainment* system a MANDATORY feature and plans to introduce an OPTIONAL *Bluetooth* feature for time point t_1.

Instead of only capturing the planned changes in natural language (e.g., as part of a specification sheet), the car manufacturer decides to construct an FM evolution plan consisting of an initial FM and an ordered sequence of scheduled edit operations. Each edit operation is scheduled to a time point that serves as intended implementation date. Arrow ⟨2⟩ in Figure 2 depicts this process, pointing to the resulting FM that is planned to be implemented for time point t_1.

The idea of an FM evolution plan is to capture planned edit operations to keep track of yet unimplemented future FM versions as abstract realization goals. The first sketch of a future FM version can be constructed in a rough, coarse-grained manner and, then, be refined with details as its assigned future date moves closer. At the same time, the FM evolution plan keeps all planned FM versions synchronized and in one model. A change that is performed on an intermediate version is automatically propagated to all subsequent versions. This offers a basis for various analyses, e.g., to ensure that critical configurations are still supported after evolution [44].

To formally define an FM evolution plan, we first define an FM as follows: An FM is a term $FM(RootFeatureID, FT)$, where $RootFeatureID$ is the ID of the root feature, and FT is a feature table that maps feature IDs to tuples. An entry within the feature table is structured as follows:

$$[FeatureID \mapsto (Name, ParentFeatureID, \overline{Groups}, FType)]$$

where $FeatureID$ is the ID of a feature, $Name$ is the name of the mapped feature, $ParentFeatureID$ is the parent feature ID, $\overline{Groups}$ is a set of child groups, and $FType$ is a variation type (i.e., OPTIONAL or MANDATORY). A group is defined as a tuple $(GroupID, GType, \overline{Features})$, where $GroupID$ is the group's ID, $GType$ is the variation type of the group (i.e., AND, OR, or ALTERNATIVE), and $\overline{Features}$ is a set of child feature IDs.

Based on this formalization of an FM, we define an FM evolution plan using a natural numbers model of time.

*Definition 3.1 (**FM Evolution Plan**).* An FM evolution plan consists of (i) an initial feature model $FM(RootFeatureID, FT)$ and (ii) an ordered list of *planning sections* containing all edit operations that are scheduled for the same time point $t \in \mathbb{N}$.

*Definition 3.2 (**Planning Section**).* A planning section within an FM evolution plan is a tuple $(t_i, \overline{operations})$, where $\overline{operations}$ is an ordered list of edit operations and $t_i \in \mathbb{N}$. All edit operations within an FM evolution plan that are associated with the same time point are mapped to one planning section. Respectively, we define the ordered list of planning sections within an FM evolution plan as $(t_1, Op_{1,1}Op_{1,2}...Op_{1,m_1}); ...; (t_n, Op_{n,1}Op_{n,2}...Op_{n,m_n})$, where each $t_i \in \mathbb{N}$ is a time point and each $Op_{i,j}$ is an edit operation.

An FM evolution plan must not contain two planning sections that are attributed to the same time point. However, additional edit operations can be associated with arbitrary existing or new planning sections. This enables integrating edit operations as intermediate planning steps in retrospect.

4 ENSURING STRUCTURAL CONSISTENCY OF FM EVOLUTION PLANS

Over the lifetime of an SPL, an FM evolution plan needs to be extended and adapted to incorporate both perspective and short-term changes. Hence, an important activity is to replan an already constructed plan by introducing new intermediate steps retroactively. However, this can introduce incompatible changes to an FM evolution plan and, thus, lead to serious inconsistencies referred to as *evolution paradoxes* [45]. A single evolution paradox can damage the structural consistency of an evolution plan to the point of making its implementation impossible so that avoiding their introduction becomes paramount. However, manually foreseeing potential effects of an intermediate edit operation on the entire evolution plan is infeasible esp. for large FMs and evolution plans with many steps. Thus, concepts for adequate FM evolution planning must guarantee structural consistency through automated and proactive prevention of evolution paradoxes.

In the following, we give an example that illustrates structurally inconsistent replanning activity before assessing how evolution paradoxes arise and classifying them into different categories. Furthermore, we introduce a formal approach that guarantees FM evolution plans to be free from paradoxes.

4.1 Challenges for Structural Consistency

Due to financial difficulties, the example car manufacturer is obliged to cut costs for their planned car SPL. In a short-notice decision, the car system is supposed to drop its stand-alone *Infotainment* system. Instead, the car manufacturer focuses on the integration of mobile phones via the *Bluetooth* feature. For the FM evolution plan, this requires replanning activity: The manufacturer retroactively plans to delete the feature *Infotainment* at the new intermediate time point $t_{0.5}$, after the initial version for t_0 and before the originally planned version for t_1. It is important to note that retroactively inserting intermediate edit operations entails specific challenges due to the different orders of devising and scheduling changes (cf. Figure 3). Hence, replanning requires two consecutive phases to maintain an overall FM evolution plan: first, the new intermediate edit operation

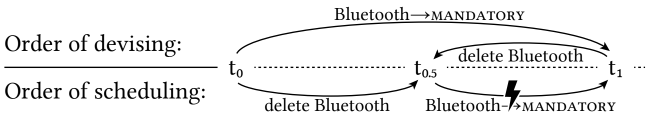


Figure 3: Difference between orders of scheduling and devising when retroactively inserting an intermediate edit operation.

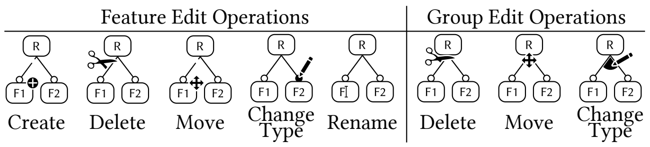


Figure 4: Common user-level edit operations for feature models.

is introduced to the evolution plan (arrow ③ in Figure 2), second, all previously devised, yet subsequently scheduled edit operations must be incorporated into the replanned state of the FM evolution plan (arrow ④). However, incorporating subsequently scheduled edit operations can fail and, thus, damage the structural consistency of an FM evolution plan: In the car example, the planned change for t_1 (add *Bluetooth* as child of *Infotainment*) can no longer be implemented as, once reaching t_1, *Infotainment* will have been deleted in the previous $t_{0.5}$. The retroactive intermediate change at $t_{0.5}$ introduced an evolution paradox into evolution plan [45].

4.2 Classification of Evolution Paradoxes

The source of every evolution paradox is an edit operation that is retroactively introduced to an FM evolution plan as a new *intermediate* step. Intermediate edit operations within an FM evolution plan change the basis for subsequently scheduled edit operations. In this way, a new intermediate edit operation can cause a subsequently scheduled edit operation to violate FM well-formedess rules (cf. *WF1*-*WF8* in Section 2) and, thus, introduce an evolution paradox.

To give a general overview of how evolution paradoxes arise, we identify a set of syntactic FM edit operations in accordance with previous work as listed in Figure 4 [47, 65]. We examine each edit operation with regard to possible violations of FM well-formedness rules to identify under which circumstances it can cause an evolution paradox. We identify four categories of evolution paradoxes.

A **Non-Existent Element Edit Paradox** arises if an evolution plan contains an edit operation on a feature/group (or its sub-tree), but this feature/group is deleted in an intermediate step (violation of *WF5* and *WF6*).

A **Variation Type Paradox** occurs if an intermediate edit operation results in an ALTERNATIVE or OR group that (i) contains a MANDATORY feature (*WF7*) or (ii) has less than two child features (*WF8*). Possible conflicting intermediate edit operations for this paradox category are:

- *feature delete operation* (*WF8*)
- *feature move operation* (*WF7* and *WF8*)
- *feature type change operation* if the type is changed to MANDATORY (*WF7*)
- *group type change operation* if the type is changed to ALTERNATIVE or OR (*WF7* and *WF8*)

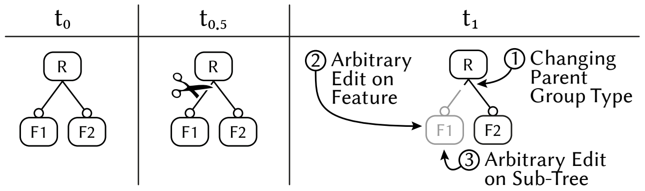

t_0
$t_{0,5}$
t_1
R
F1
F2
② Arbitrary Edit on Feature
① Changing Parent Group Type
③ Arbitrary Edit on Sub-Tree

Figure 5: Graphical representation of FM edit operations that might conflict with an intermediate *feature delete operation.* This figure is an excerpt from our online appendix[1].

A **Naming Conflict Paradox** is caused by an intermediate *feature rename operation* that changes a feature's name so that the new name is not unique within at least one subsequently planned FM version (*WF3*).

A **Transient Effect Paradox** can be caused by an intermediate edit operation that becomes unintentionally ineffective and, thus, limited in its effect by a subsequent operation in the immediate future. Possible intermediate edit operations are:

- *feature create operation* - can become unintentionally ineffective due to a subsequently scheduled delete operation targeting the created feature's parent structure.
- *feature/group move operation* - can become unintentionally ineffective by a subsequently scheduled (i) move operation that again relocates the moved feature/group or (ii) delete operation targeting the new parent structure.
- *feature/group type change operation* - a type change can become unintentionally reverted by a subsequently scheduled type change on the same feature/group.
- *feature rename operation* - the effect of renaming a feature can become unintentionally ineffective by a subsequently scheduled delete operation on the renamed feature.

In our online appendix[1], we provide further, detailed information about the exact circumstances that result in an evolution paradox, broken down by each individual kind of FM edit operation. As an excerpt, Figure 5 shows how an intermediate *feature delete operation* can cause various evolution paradoxes. Arrows ①, ② and ③ represent operations in a subsequent state of the FM evolution plan that conflict with the intermediate feature delete operation:

① *Variation Type Paradox* - A previously defined type change on the parent group to either ALTERNATIVE or OR in t_1 can cause a violation of *WF8* if only one other child feature remains.

② *Non-Existent Element Edit Paradoxes* - Previously defined but subsequently scheduled edit operations on feature *F1* in t_1 (such as renaming or moving the feature) cannot be executed, as *F1* is deleted by the intermediate feature delete operation.

③ *Non-Existent Element Edit Paradoxes* - Similarly, previously defined but subsequently scheduled operations on an arbitrary element in the sub-tree of *F1* in t_1 cannot be executed as *F1* (along with its sub-tree) are deleted in $t_{0.5}$.

[1]https://gitlab.com/Adomat/consistent-feature-model-evolution-plans

4.3 Paradox-Free Execution Semantics for Evolution Plans

In the following, we describe an execution semantics for FM evolution plans that is able to verify the structural consistency of an FM evolution plan. We check a modified FM evolution before new intermediate edit operations become effective and, thus, guarantee FM evolution plans to be free from evolution paradoxes through prohibiting operations that would introduce structural inconsistencies. For that purpose, we define the state and the behavior of an FM evolution plan and, based on that, use rewriting logic to check whether the execution of each contained edit operation would violate any FM well-formedness rules (cf. Section 2).

4.3.1 State of an FM evolution plan. We refer to the planning section that contains a currently inspected edit operation as the *active planning section.*

*Definition 4.1 (**Active Planning Section**).* We define the active planning section as $ActiveSection = (FM(RootFeatureID, FT)\ Op_{cur,1}...Op_{cur,m_{cur}})$ with t_{cur} being the current time. At most one planning section within an FM evolution plan can be active at a given time $t_{cur} \in \mathbb{N}$.

Using above definition, a concrete state of an FM evolution plan is defined by a combination of the current time, an active planning section and a list of remaining sections.

*Definition 4.2 (**Planning State**).* A *planning state* is defined as a tuple $(t_{cur}, ActiveSection, \overline{RemainingSections})$, where $t_{cur} \in \mathbb{N}$ is the current time, *ActiveSection* is the currently active planning section, and $\overline{RemainingSections}$ is an ordered list of all remaining planning sections for subsequent time points.

4.3.2 Behavior of an FM evolution plan. We use structural operational semantics (SOS) to specify the behavior of FM edit operations. The complete semantics for all FM edit operations listed in Figure 4 can be found in our online appendix[1]. As an example, the execution of a feature create operation is defined as follows:

SEMANTICS RULE 4.1 (***Feature Create Operation***).

$$\frac{\begin{array}{c}(1)\ FT(FeatureID) = \perp \\ (2)\ isUniqueName(Name, FT) \\ (3)\ FT' = addFeatureToGroup(FT, ParentGroupID, FeatureID) \\ FT'' = FT' + [FeatureID \mapsto (Name, TargetFid, \varnothing, FType)] \\ (4)\ isValidType(FT'', FeatureID)\end{array}}{\begin{array}{c}FM(RootFeatureID, FT) \\ \textbf{\textit{createFeature}}(FeatureID, Name, ParentGroupID, FType) \\ \Rightarrow \\ FM(RootFeatureID, FT'')\end{array}}$$

SEMANTICS RULE 4.1 defines the semantics of a feature create operation so that the operation is applicable iff (above the line):

(1) the given *FeatureID* is not yet assigned to another feature,
(2) no other feature is assigned the name *Name* (*WF3*),
(3) the parent group with ID *ParentGroupID* exists (*WF5*),
(4) the given variation type *FType* of the new feature is valid within the resulting FM (*WF3*, *WF7* and *WF8*)

We define helping functions "*isUniqueName*", "*addFeatureToGroup*" and "*isValidType*". If all conditions are met, the effect of executing the operation is applied (below the line). The group with ID *ParentGroupID* is extended by a child feature entry and a new feature mapping $[FeatureID \mapsto (Name, TargetFid, \emptyset, FType)]$ is added to the feature table. The empty set argument constructs the new feature with an empty set of child group IDs.

Based on these semantics of FM edit operations, we construct a paradox-free execution semantics for FM evolution plans. It consists of three rules that are applied depending on the planning state of the inspected FM evolution plan:

*Semantics Rule 4.2.1 (**Executing FM Edit Operations**).*

$$\frac{\begin{array}{c}FM(RootFeatureID, FT)\ Op_{cur,1} \\ \Rightarrow \\ FM(RootFeatureID, FT')\end{array}}{\begin{array}{c}(t_{cur}, FM(RootFeatureID, FT)\ Op_{cur,1}...Op_{cur,m_{cur}}, \overline{RemainingSections}) \\ \Rightarrow \\ (t_{cur}, FM(RootFeatureID, FT')\ Op_{cur,2}...Op_{cur,m_{cur}}, \overline{RemainingSections})\end{array}}$$

Semantics Rule 4.2.1 is applied if the planning state's active planning section still contains FM edit operations. It checks whether the next FM edit operation to be applied ($Op_{cur,1}$) is free from structural inconsistencies (using its semantics). If (and only if) this is the case, it is removed from the active planning section and the feature table is modified from FT to FT'. We denote an empty list of remaining edit operations within the active planning section as ϵ.

*Semantics Rule 4.2.2 (**Advancing Time**).*

$$\frac{t_{cur} < t_{next}}{\begin{array}{c}(t_{cur}, FM(RootFeatureID, FT)\ \epsilon, (t_{next}, \overline{Ops}); \overline{MoreSections}) \\ \Rightarrow \\ (t_{next}, FM(RootFeatureID, FT)\ \overline{Ops}, \overline{MoreSections})\end{array}}$$

Semantics Rule 4.2.2 advances the current time t_{cur} to t_{next} if no edit operation is left in the active planning section. The subsequent planning section is then set as active planning section.

*Semantics Rule 4.2.3 (**Fully Processed Plan**).*

$$\frac{}{(t, FM(RootFeatureID, FT)\ \epsilon, \rho)}$$

Semantics Rule 4.2.3 defines the successful execution of the entire FM evolution plan iff all edit operations within all planning sections were applied successfully. Note that we use ρ to denote an empty list of remaining planning sections within a planning state.

We formulate the theorem that the above execution semantics ensures the structural consistency of an FM evolution plan.

Theorem 4.3 (**Structural Consistency of an FM Evolution Plan**). Let *Plan* be an FM evolution plan with an initial $FM(RootFeatureID, FT)$ and a non-empty list of planning sections $(t_1, \overline{Ops_1}); (t_2, \overline{Ops_2}); ...; (t_n, \overline{Ops_n})$. Then, *Plan* is structurally consistent iff the execution of all planning sections on $FM(RootFeatureID, FT)$ according to rules 4.2.1, 4.2.2, and 4.2.3 terminates:

$$\begin{array}{l}(t_0,\ FM(RootFeatureID, FT)\ \epsilon,\ Plan) \\ \overset{*}{\Rightarrow} (t_n,\ FM(RootFeatureID, FT')\ \epsilon,\ \rho)\end{array}$$

5 ENSURING PROJECT-SPECIFIC LOGICAL CONSISTENCY OF EVOLUTION PLANS

In the previous section, we provide concepts that enable insertion of planned edit operations at arbitrary time points while guaranteeing *structural* consistency of an FM evolution plan. However, this alone is not sufficient for ensuring the *logical* consistency of an FM evolution plan. Planning evolution by means of edit operations requires to know (a) what exact edit operations are supposed to be scheduled and (b) the degree of freedom in replanning. Especially for changes planned far in the future, determining an exact time point for their realization is often infeasible. Nevertheless, project-specific concerns may govern potential scheduling orders, e.g., when a planned change depends on another planned change that has to be scheduled before it. These considerations are essential for determining a tentative time point for scheduling a change but, even more so, to determine the degree of freedom for replanning while still maintaining logical consistency within the resulting evolution plan. Suitable evolution planning must include support for such project-specific concerns. While respective considerations exist in various forms for general project management [7, 64], at present, SPL engineering has widely neglected these challenges. In the following, we describe a situation that requires this additional flexibility before we present concepts that address the resulting challenges.

5.1 Challenges for Logical Consistency

The car manufacturer decides to extend the variability of the *Sensors* functionality. For that purpose, a distinction between *Front Sensors* and *Rear Sensors* is supposed to enable customers to configure their assistance system on a more fine-grained level. However, the exact time point for an implementation cannot yet be scheduled as it depends on unforeseeable delivery conditions with a third-party company. An overhasty decision for a concrete time point can result in a too tightly or too loosely scheduled evolution plan. In the worst case, this entails revoking and reapplying all replanned changes across different FM versions, causing additional effort and cost. Hence, it is important to the car manufacturer's business strategy to leave this decision open. Nevertheless, it is crucial to capture all kinds of planned changes as it needs to be ensured that every part of the plan is eventually implemented. The car manufacturer sets a time interval in which the planned change is supposed to be implemented, i.e., at some time point before t_2, but not sooner than the introduction date of *Sensors*. The FM evolution plan needs to be extended in terms of adding *Front Sensors* and *Rear Sensors* as children of *Sensors*. This change is supposed to be implemented at some point within the right-open time interval [*Sensors* was introduced, t_2) as illustrated at the bottom of Figure 2. The time interval's start date is not fixed to a specific time due to the dependence on *Sensors* and because the introduction of *Sensors* could possibly be rescheduled to another time point than t_0.

5.2 Feature Model Evolution Constraints

To incorporate planned changes into an FM evolution plan based on possible time intervals and dependencies on the scheduling of other changes, we devise a concept we call *feature model evolution constraints* (FMECs). In the following, we first introduce core concepts and the syntax of FMECs and, then, define the semantics of FMECs via a translation to LTL.

5.2.1 Syntax and core concepts of FMECs. The complete grammar of the syntax for FMECs can be accessed via our online appendix[1]. While our formalized FM evolution plans (cf. Definition 3.1) use a natural numbers model of time, note that FMECs enable to work with actual dates. In the following, we summarize the core concepts of FMECs along with examples of their syntax:

An **Evolutionary Property** of an FM evolution plan is:

- the presence of a referenced feature or group:
  `Sensors valid`
- the presence of a certain variation type of a referenced feature or group:
  `Infotainment.type == MANDATORY valid`
- the validity of a parent-child-relationship between (i) two features or (ii) a feature and a group (or vice versa):
  `Front Sensors.parentFeature==Sensors valid`

A Feature or group can be referenced either by name (if it is unique within the evolution plan) or by its ID. Evolutionary properties (except feature/group presences) can be valid in multiple time points which are not necessarily connected.

An **Evolutionary Event** allows to express the start or end of a feature/group presence. We assume that each feature/group can only be created/deleted once in an FM evolution plan.
`Bluetooth starts` or `Sensors ends`
Evolutionary events are unique within an FM evolution plan.

Evolutionary properties and evolutionary events are the basic building blocks to construct two kinds of FMECs:

A **Time Point Constraint** puts an evolutionary property or event in relation to a time point, using one of the keywords before, until, after, when or at. A time point can be either explicit (i.e., a concrete date) or implicit in terms of another evolutionary property or event. It is possible to specify a temporal offset to or from the referenced time point, e.g.:

```
Front Sensors starts before
    Sensors starts - 2 months
```

A **Time Interval Constraint** puts an evolutionary property or event in relation to a right-open time interval, where both interval boundaries are (implicit or explicit) time points, e.g.:

```
Front Sensors starts during
    [ Sensors starts , 21.09.2024 12:00 )
```

Two FMECs can be connected using the boolean connectives **and** (conjunction), **or** (disjunction), **implies** (implication), and **not** (negation) resulting in a new FMEC. Additionally, FMECs can be nested using parentheses.

5.2.2 Semantics of FMECs. We define the semantics of FMECs through a translation to linear temporal logic (LTL) as shown in Table 2. Negations, conjunctions, disjunctions and implications within FMECs are translated to LTL using the corresponding logical operators. With a translation to LTL formulae according to Table 2, we can verify whether FMECs hold over the model structure as defined by a formalized FM evolution plan according to the definitions in Section 3. We use the Maude LTL model checker to evaluate translated FMECs. The model checker uses an implementation of our execution semantics as its linear state model. A branching temporal logic is not necessary as all FM edit operations behave deterministically. We define helping functions that evaluate whether the currently active time point equals (isTime(t_i)) or is greater (after(t_i)) than the input time point $t_i \in \mathbb{N}$.

We evaluate each FMEC within an FMEC module separately. An FM evolution plan is logical consistent in regards to its specified FMECs if all FMECs are evaluated positively.

6 EVALUATION

In this paper, we present a concept that ensures the structural and logical consistency of an FM evolution plan. This is achieved by (i) guaranteeing an FM plan to be free from evolution paradoxes and (ii) verifying the satisfaction of additional, project-specific FMECs. In this section, we evaluate our method in terms of its feasibility and performance. We show the feasibility of our method by implementing and applying it to real-world FM evolution. To measure performance, we conduct a series of in-depth experiments.

6.1 Evaluation Setup

We implement our method by integrating it into the evolution-aware FM tool suite DarwinSPL[2] [41]. DarwinSPL enables to construct and modify FM evolution plans, i.e., it keeps track of scheduled edit operations as well as the resulting evolution-aware FM. We extend DarwinSPL with (i) an editor and parser for FMEC expressions and (ii) functionality to translate evolution plans and FMECs to our formalized notation (cf. Sections 3, 4 and 5). We further use the Maude system [13] to realize our method in rewriting logic based on our formalization and connect it with DarwinSPL. The complete implementation can be found in our online appendix[1].

To gain input data for this evaluation, we gather several existing FM evolution scenarios that we use as FM evolution plans. For one, we make use of previously collected SPL evolution histories for four small to medium-sized, artificial, yet, well documented FM evolution scenarios from different domains [38, 39]: *Mine Pump*, *Wiper*, *Vending Machine* and *Body Comfort System*. In addition, we use the real-world scenario *FinancialServices01* from the *FeatureIDE* [35] repository to evaluate our concepts on a large-scale plan.

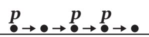
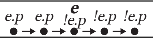
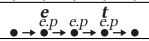
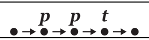
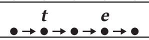
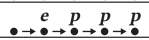
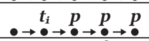
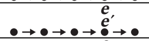
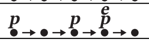
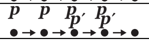
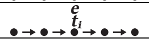
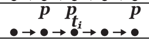
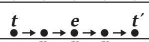
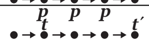
We reconstruct all gathered FM evolution scenarios within the DarwinSPL tool suite to use them in combination with our Maude implementation. For the evolution histories of Mine Pump, Wiper, Vending Machine and Body Comfort System, we perform this task manually, i.e., by capturing the initial state of each evolution scenario from their documentation before successively remodeling all following planning sections. We convert the large-scale evolution history *FinancialServices01* from a set of different FeatureIDE models, each representing a different version within the FM evolution scenario, to the evolution-aware DarwinSPL notation with the use of DarwinSPL's import mechanism. Table 3 shows properties of all resulting evolution-aware DarwinSPL FMs.

6.2 Evaluation of Evolution Paradox Prevention

We define our FM evolution plan execution semantics in form of inference rules and implement these in rewriting logic. Our rules ensure a correct application of edit operations with respect to well-formedness rules *WF1-WF8*. In consequence, an FM evolution plan is guaranteed to be free of evolution paradoxes (cf. Theorem 4.3) with a terminating execution of our semantics.

[2]https://gitlab.com/DarwinSPL/DarwinSPL

Table 2: Translation of FMECs to LTL.

FMEC concept	Keyword	FMEC expression	Visualization	LTL	Notes
Evolutionary Property	valid	<Property>		-	N_p
Evolutionary Event	starts/ends	<FeatureID/GroupID> starts/ends		-	$N_{e.p}$
Time Point Constraint	before	<Event> before <TP>		$\Diamond(e \wedge (\bigcirc \Diamond t) \wedge (e.p\ U\ t))$	$N_{e.p}$, N_t
	until	<Property> until <TP>		$\Box(p \rightarrow (p\ U\ t)) \wedge \Diamond p$	N_p, N_t
	after	<Event> after <TP>		$\Diamond(t \wedge \bigcirc \Diamond e)$	N_t
		<Property> after <Event>		$\Diamond(e \wedge \bigcirc \Box p)$	N_p
		<Property> after <ExplTP>		$\Box(\text{after}(t_i) \rightarrow p)$	N_p
	when	<Event> when <Event'>		$\Diamond(e \wedge e')$	
		<Event> when <Property>		$\Diamond(e \wedge p)$	N_p
		<Property> when <Property'>		$\Box(p' \rightarrow p)$	N_p
	at	<Event> at <ExplTP>		$\Diamond(\text{isTime}(t_i) \wedge e)$	
		<Property> at <ExplTP>		$\Diamond(\text{isTime}(t_i) \wedge p)$	N_p
Time Interval Constraint	during	<Event> during [<TP> , <TP'>]		$\Diamond(t \wedge \Diamond(e \wedge \bigcirc \Diamond t'))$	N_t
		<Property> during [<TP> , <TP'>]		$\Diamond(t \wedge (p\ \text{U}\ t'))$	N_p, N_t

N_p: We assume an evolutionary property to be evaluated to FALSE for time points in which its referenced feature or group does not exist. For instance, "*Sensors*.type==MANDATORY valid" is FALSE for time points in which *Sensors* does not exist.

$N_{e.p}$: We denote the evolutionary property of an evolutionary event e with $e.p$. For instance, e = "*Sensors* starts" with $e.p$ = "*Sensors* valid".

N_t: We use "t" in our LTL formulae ("<TP>" in FMECs) to be substituted with either an explicit time point "isTime(t_i)" or an event "e".

Table 3: Properties of the used FM evolution scenarios.

	Overall Features	Overall Groups	Planning Sections	Edit Operations
MinePump	9	3	3	25
Wiper	14	5	4	42
Vend. M.	19	7	6	72
BodyComfS	48	16	5	144
FinServices01	1,083	259	10	2,370

In the following, we evaluate the performance of our paradox-free FM evolution plan execution semantics. To prevent the introduction of structural inconsistencies, an FM evolution plan needs to be inspected whenever new intermediate changes are integrated, e.g., upon saving the modified evolution plan. To avoid a negative impact on the usability of FM tool suites that integrate our means, it is crucial to achieve a performance acceptable for productive work.

In the following, we measure the performance of our Maude implementation for the paradox-free FM evolution plan execution semantics and show that it scales even for large evolution histories. To simulate individual edit operations by users, we generate a random edit operation that we insert as intermediate step into one of the FM evolution plans from our collection listed in Table 3. We then use this randomly extended FM evolution plan to run our execution semantics while measuring the runtime. For each FM evolution plan, we repeat this process 100,000 times.

Figure 6 depicts the distribution of respective runtimes for each evolution scenario. For the sake of better readability, we omit outliers. Table 4 summarizes detailed information in terms of average and maximum runtimes for each evolution scenario. On average, our execution semantics is able to analyze the structural consistency of the four small to medium-sized FM evolution scenarios in less than 40 ms. For the large FM evolution scenario *FinancialServices01*, this task requires an average runtime of 692 ms. These results show that our implementation is able to analyze even complex FM evolution plans within a short period of time and, thus, that it is suitable for productive application, e.g., when performing them on saving an evolution plan. With regard to FM evolution plans that are more complex than *FinancialServices01*, we point out that it is sufficient to perform checks on the structural consistency of a plan upon saving it (as opposed to checking each operation).

Figure 7 gives an overview of the average runtime of our Maude implementation for the *FinancialServices01* scenario, categorized by the kind of edit operation that is inserted as intermediate planning step. With a maximum difference of 63.9 ms, we observe that the runtime of our execution semantics is stable against the kind of edit operation that is inserted into an FM evolution plan. Considering the significant size of the FM and evolution plan, we deem the respective runtimes as adequate for practical application.

Furthermore, Table 4 contains information on the percentage of randomly generated intermediate operations that would cause the introduction of an evolution paradox if applied. We observe that this is the case for 23%-37% of the randomly generated intermediate edit operations. Taking into account that each of these paradoxical edit operations would violate the structural consistency of the evolution plan, we see this additional finding as a confirmation for the urgence of automated detection mechanisms within FM evolution planning tools.

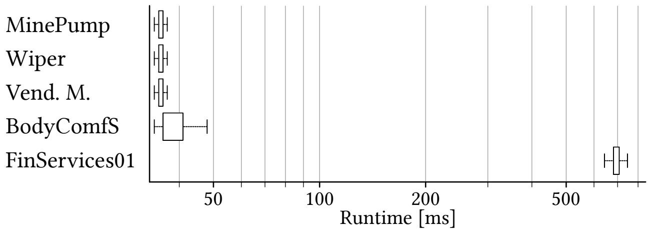


Figure 6: Runtimes of our execution semantics over 100,000 test runs for each scenario (outliers omitted). In each test run, we insert one randomly generated intermediate edit operation.

Table 4: Detailed information from analyzing 100,000 randomly generated intermediate edit operations for each scenario.

	Percentage of Paradoxical Operations	Average Maude Runtime [ms]	Maximum Maude Runtime [ms]
MinePump	23.49%	36	249
Wiper	36.75%	36	254
Vend. M.	31.01%	36	247
BodyComfS	24.60%	38	255
FinServices01	36.17%	692	2764

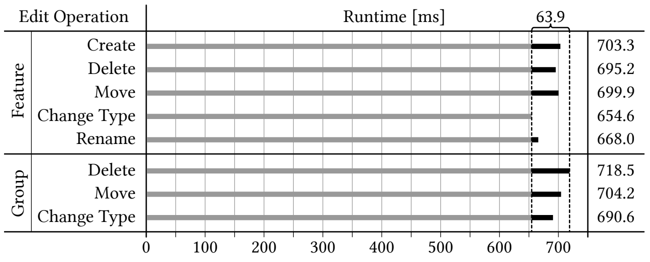


Figure 7: Average runtime of our execution semantics for the *FinancialServices01* scenario, categorized by the kind of edit operation that was inserted as intermediate planning step.

6.3 Evaluation of FMEC Verification

We verify FMECs by evaluating their LTL translation using the Maude LTL model checker. We let the model checker work on the basis of our rewriting logic implementation for the execution semantics, which creates a linear state model. For practical relevance, this process must scale with large FM evolution plans. Thus, we create 12 FMECs for each of the input evolution scenarios, covering all of our translation rules shown in Table 2. The entire collection of constructed FMECs can be accessed via our online appendix[1].

We start by measuring the average runtime for the verification of a single FMEC over 500 repetitions for each evolution scenario. We then repeat this procedure while gradually increasing the number of FMECs verified simultaneously up to a total number of 12. Figure 8 gives an overview of our results. We compare the FMEC verification runtimes to the average runtimes of our execution semantics for the underlying paradox detection (cf. Figure 6) and observe that, even

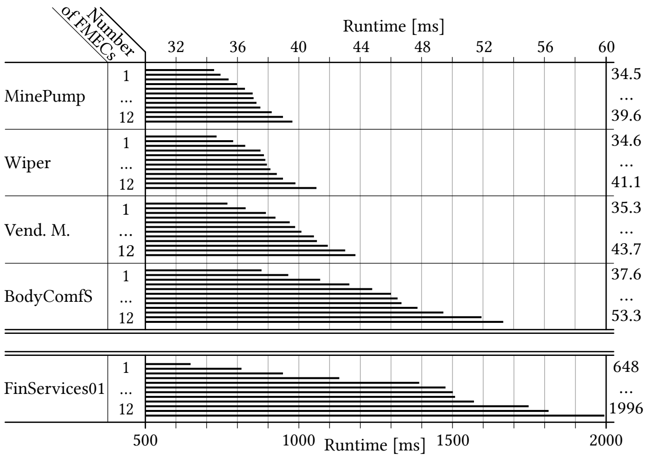


Figure 8: Runtime of our execution semantics when checking 1, 2, ... 12 FMECs simultaneously. Each bar depicts an average value over 500 tests. Please note the differing scale for *FinancialServices01*.

though each additional FMEC constitutes a measurable increase in computation time, it does not lead to excessive delays. For the four small to medium-sized evolution scenarios, each additional FMEC causes merely a few milliseconds of additional computation time. For the *FinancialServices01* scenario, we observe a more noticeable increase in computation time: While a single FMEC is verified with an average runtime of 648 ms, the simultaneous verification of 12 FMECs requires an average runtime of 1996 ms. However, ranging within the time span of only a few seconds, we argue that our execution semantics are well suitable for evolution planning. An important factor for this conclusion is that (i) the verification of a collection of FMECs can be split in arbitrarily large FMEC modules and (ii) the verification of FMECs is not necessarily performed as frequently as an underlying paradox detection mechanism but rather on user demand.

Our evaluation shows that the implementation of our execution semantics based on Maude is able to ensure the structural and logical consistency of an FM evolution plan with acceptable runtimes. An integration of these means into FM tool suites allows for sophisticated SPL planning on FM level.

6.4 Threats to Validity

Our execution semantics check FM well-formedness rules using SOS and an external rewriting logic. We use existing well-formedness rules from the literature and an existing rewriting logic framework (Maude) that both have been used in various other publications. This leaves as internal threat to validity that we might have wrongly encoded some of the well-formedness rules in our semantics rules. However, the rest of our method is correct by definition.

The FM evolution scenarios used for this evaluation are examples from other research projects and/or real-world projects. Thus, we argue that they are not biased towards suiting our solutions. However, the scenarios were reconstructed from results of already performed evolution. As a consequence, they contain only the least number of edit operations necessary to remodel each evolution scenario. This leads to an uneven distribution of edit operations, e.g., for the *FinancialServices01* evolution scenario with 9,639 feature variation type change operations and 0 group move operations.

Regarding the random generation of intermediate edit operations in Section 6.2, we see as internal threat to validity whether this procedure simulates real-world user behavior.

Furthermore, the reconstruction of *FinancialServices01* within DarwinSPL relies on the tool's import mechanism. To reduce the risk of falsely imported evolution scenarios, we check each resulting evolution-aware feature model for sanity by comparing each planning section with the respective FeatureIDE source model.

7 RELATED WORK

SPL evolution is a widely investigated area [9, 23, 32] with work on modeling and performing evolutionary modifications to an FM [22, 40, 54, 59], its associated realization artifacts [28, 50] or both [27, 56, 60]. A range of tools tackles selected areas of SPL evolution, e.g., *DeltaEcore* [62], *ECCO* [14], *SuperMod* [57, 58], *VaVe* [4], *SiPL* [49], *EvoFM* [10, 11], *EvoPL* [9, 55] or *DarwinSPL* [41]. Recently, there have also been first attempts to harmonize modeling of SPL evolution by integrating [17] or abstracting from [3] various of the different underlying notations. Even though the need and principle feasibility for SPL evolution planning has been recognized before [9, 48], the vast majority of existing concepts and tools does not tackle this challenge. Nevertheless, we deem selected evolution-aware concepts of individual approaches relevant to the work of this paper so that we discuss them below.

The field of conflict and dependency analysis for graph transformation systems (GTS) put forth methods for a detection of incompatibilities between general graph modifications [5, 24, 25, 30]. Such methods could be utilized to prevent incompatibilities between operations within an FM evolution plan and, thus, ensure its structural consistency. However, as our work additionally lays focus on the logical consistency of plans, we devise an integrated solution on the basis of SOS and LTL to ensure both logical and structural consistency, custom-tailored for FM evolution planning.

Ample work defines modification operations on FMs on various levels of granularity [22, 40, 54, 59]. The FM operations we define manipulate the atomic characteristics of an FM so that we argue that more complex operations, as in some of these approaches, can be composed from our operations. We intend to add explicit support for FM evolution planning with complex operations as future work.

Various approaches provide retroactive support for FM evolution operations by extracting them from different states of a feature model: In particular, *Kehrer at al.* introduce *SiLift* to determine model change operations from different model states [21] and provide this functionality for SPLs as part of the tool *SiPL* [49]. Moreover, *Bürdek et al.* gain knowledge about user-level edit operations on FM evolution histories by means of a differencing algorithm between FM versions that also takes semantic differences into account [12]. However, when planning FM evolution, operations have not yet been performed so that these methods are not directly applicable in our context. Nevertheless, we intend to investigate them further to provide an import mechanism to our concepts when FM evolution was planned by explicitly modeling separate FM states as is the case with some of our industry partners.

A number of FM notations exists that integrate concerns of evolution: *Mitschke and Eichberg* introduce a concept of feature-driven versioning that allows to annotate features and realization artifacts with versioning numbers [37]. While their notation enables to model that a feature model was changed, it does not explicitly store how it changed. *Seidl et al.* introduce *hyper-feature models* (HFMs) as an FM notation that augments features with version numbers where each feature version is mapped to realization artifacts in the respective implementation version as part of the tool suite *DeltaEcore* [61–63]. Their notation can model changes to associated realization artifacts on FM level but not FM evolution itself. *Hinterreiter et al.* present a mechanism that keeps track of different versions of an FM by means of the variability-aware version control system *ECCO* [18]. This allows to capture an FM and its evolution as individual snapshots after evolution was performed but not as part of planning. *Botterweck et al.* introduce *EvoFM* as concepts to model evolution of an FM in terms of change fragments that describe the difference between one FM version to another [10, 11, 55]. This principally allows to represent past and future FM evolution but they provide neither execution semantics nor tooling as we do. *Nieke et al.* present *temporal feature models* (TFMs) as a notation that can capture changes to an FM's structure as integrated first-level entities and, based on that, *DarwinSPL* as an evolution-aware FM tool suite [44]. Their notation allows to both capture past FM changes, which have already been realized, and model future FM changes, which yet have to be performed.

The tool suite DarwinSPL is most closely related to our work as it allows to capture and, principally, plan SPL evolution on basis of the aforementioned TFMs [41]. In addition, it provides advanced functionality for anomaly analysis during evolution [42] and modeling context-awareness as an additional factor for configurations of an SPL [33, 34, 43]. Even though our concept is independent of a particular technology, it aligns well with TFMs as basis for planning of future evolution. Hence, we integrate our implementation within DarwinSPL and extend it with our method for structurally and logically consistent FM evolution planning and replanning.

8 CONCLUSION AND FUTURE WORK

With this work, we provide a solution to the as of yet unaddressed challenge of supporting FM evolution planning while ensuring consistent FM evolution plans. We split this challenge in two parts, i.e., ensuring the (i) structural and (ii) logical consistency of an FM evolution plan, and provide an integrated solution. Our concept is found in formal methods and, thus, independent of specific FM tooling. We provide an implementation using Maude [13] and an integration into the evolution-aware tool suite DarwinSPL [41]. Our evaluation shows that our concept and implementation are applicable even for large-scale real-world FM evolution.

In the future, we plan to perform a longitudinal study with our industry partner where we use our method to plan FM evolution for an extended period of time.

ACKNOWLEDGMENTS

This work is supported by the project *LTEP: Long-Term Evolution Planning for Highly Variable Software Systems* funded by the German DAAD and the Norwegian NFR.

REFERENCES

[1] 2000. Rewriting Techniques and Applications, 11th International Conference, RTA 2000, Norwich, UK, July 10-12, 2000, Proceedings *(Lecture Notes in Computer Science)*, Leo Bachmair (Ed.), Vol. 1833. Springer. https://doi.org/10.1007/10721975

[2] Luca Aceto, Wan Fokkink, and Chris Verhoef. 2001. Structural operational semantics. In *Handbook of process algebra*. Elsevier, 197–292.

[3] Sofia Ananieva, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, S. Ramesh, Andreas Burger, Gabriele Taentzer, and Bernhard Westfechtel. 2019. Towards a conceptual model for unifying variability in space and time. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*, Carlos Cetina, Oscar Díaz, Laurence Duchien, Marianne Huchard, Rick Rabiser, Camille Salinesi, Christoph Seidl, Xhevahire Tërnava, Leopoldo Teixeira, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 67:1–67:5. https://doi.org/10.1145/3307630.3342412

[4] Sofia Ananieva, Heiko Klare, Erik Burger, and Ralf H. Reussner. 2018. Variants and Versions Management for Models with Integrated Consistency Preservation. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7-9, 2018*, Rafael Capilla, Malte Lochau, and Lidia Fuentes (Eds.). ACM, 3–10. https://doi.org/10.1145/3168365.3168377

[5] Guilherme Grochau Azzi, Andrea Corradini, and Leila Ribeiro. 2018. On the essence and initiality of conflicts. In *International Conference on Graph Transformation*. Springer, 99–117.

[6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines*, Henk Obbink and Klaus Pohl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–20.

[7] John H Blackstone. 2001. Theory of constraints-a status report. (2001).

[8] G. Bockle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid. 2004. Calculating ROI for software product lines. *IEEE Software* 21, 3, 23–31. https://doi.org/10.1109/MS.2004.1293069

[9] Goetz Botterweck and Andreas Pleuss. 2014. *Evolution of Software Product Lines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 265–295. https://doi.org/10.1007/978-3-642-45398-4_9

[10] 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* (Cape Town, South Africa) *(PLEASE '10)*. Association for Computing Machinery, New York, NY, USA, 24–31. https://doi.org/10.1145/1808937.1808941

[11] 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* (Denver, Colorado, USA) *(FOSD '09)*. Association for Computing Machinery, New York, NY, USA, 109–116. https://doi.org/10.1145/1629716.1629737

[12] 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* 23, 4, 687–733. https://doi.org/10.1007/s10515-015-0185-3

[13] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (Eds.). 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, Vol. 4350. Springer. https://doi.org/10.1007/978-3-540-71999-1

[14] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *37th International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 665–668. https://doi.org/10.1109/ICSE.2015.218

[15] W. B. Frakes and Kyo Kang. 2005. Software reuse research: status and future. *Transactions on Software Engineering* 31, 7 (July 2005), 529–536. https://doi.org/10.1109/TSE.2005.85

[16] Rohit Gheyi, Tiago Massoni, and Paulo Borba. 2006. A theory for feature models in alloy. In *First alloy workshop*. Citeseer, 71–80.

[17] 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 International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*, Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm (Eds.). ACM, 115–128. https://doi.org/10.1145/3357765.3359515

[18] 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 *23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. IEEE, 107–114.

[19] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, USA.

[20] Jing Sun, Hongyu Zhang, Yuan Fang, and Li Hai Wang. 2005. Formal semantics and verification for feature modeling. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS'05)*. 303–312. https://doi.org/10.1109/ICECCS.2005.48

[21] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. 2012. Understanding model evolution through semantically lifting model differences with SiLift. In *2012 28th International Conference on Software Maintenance (ICSM)*. 638–641.

[22] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake. 2019. Foundations of collaborative, real-time feature modeling. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 36:1–36:8. https://doi.org/10.1145/3336294.3336308

[23] Miguel A Laguna and Yania Crespo. 2013. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming* 78, 8 (2013), 1010–1034.

[24] Leen Lambers, Kristopher Born, Jens Kosiol, Daniel Strüber, and Gabriele Taentzer. 2019. Granularity of conflicts and dependencies in graph transformation systems: A two-dimensional approach. *Journal of logical and algebraic methods in programming* 103 (2019), 105–129.

[25] Leen Lambers, Daniel Strüber, Gabriele Taentzer, Kristopher Born, and Jevgenij Huebert. 2018. Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In *Proceedings of the 40th International Conference on Software Engineering*. 716–727.

[26] M. M. Lehman. 1996. Laws of software evolution revisited. In *Software Process Technology*, Carlo Montangero (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 108–124.

[27] Lukas Linsbauer, Florian Angerer, Paul Grünbacher, Daniela Lettner, Herbert Prähofer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2014. Recovering Feature-to-Code Mappings in Mixed-Variability Software Systems. In *30th International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 426–430. https://doi.org/10.1109/ICSME.2014.67

[28] Sascha Lity, Sophia Nahrendorf, Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2018. 175% Modeling for Product-Line Evolution of Domain Artifacts. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7-9, 2018*, Rafael Capilla, Malte Lochau, and Lidia Fuentes (Eds.). ACM, 27–34. https://doi.org/10.1145/3168365.3168369

[29] Neil Loughran, Awais Rashid, Weishan Zhang, and Stan Jarzabek. 2004. Supporting product line evolution with framed aspects. In *AOSD ACP4IS Workshop*. Citeseer.

[30] Rodrigo Machado, Leila Ribeiro, and Reiko Heckel. 2015. Characterizing Conflicts Between Rule Application and Rule Evolution in Graph Transformation Systems. In *International Conference on Graph Transformation*. Springer, 171–186.

[31] Zohar Manna and Amir Pnueli. 1992. *The temporal logic of reactive and concurrent systems - specification*. Springer. https://doi.org/10.1007/978-1-4612-0931-7

[32] Maíra Marques, Jocelyn Simmonds, Pedro O. Rossel, and María Cecilia Bastarrica. 2019. Software product line evolution: A systematic literature review. *Information and Software Technology* 105 (2019), 190 – 208. https://doi.org/10.1016/j.infsof.2018.08.014

[33] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2016. Context aware reconfiguration in software product lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. 41–48.

[34] 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.

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

[36] José Meseguer. 2012. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming* 81, 7-8 (2012), 721–781. https://doi.org/10.1016/j.jlap.2012.06.003

[37] Ralf Mitschke and Michael Eichberg. 2008. Supporting the evolution of software product lines. In *ECMDA Traceability Workshop (ECMDA-TW)*. Citeseer, 87–96.

[38] Sophia Nahrendorf. 2017. Entwicklung und Modellierung von Evolutionsszenarien für Delta-orientierte Softwareproduktlinien: Projektarbeit. https://doi.org/10.24355/dbbs.084-201704071225

[39] Sophia Nahrendorf. 2017. *Integration von Evolution in die Modellierung und Analyse von Softwareproduktlinien*. Master's thesis. Braunschweig. https://doi.org/10.24355/dbbs.084-201711071415

[40] 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 International Conference on Generative Programming and Component Engineering*. 33–42.

[41] 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* (Eindhoven, Netherlands) *(VAMOS '17)*. Association for Computing Machinery, New York, NY, USA, 92–99. https://doi.org/10.1145/3023956.3023962

[42] 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 International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 188–201. https://doi.org/10.1145/3278122.3278123

[43] Michael Nieke, Jacopo Mauro, Christoph Seidl, and Ingrid Chieh Yu. 2016. User profiles for context-aware reconfiguration in software product lines. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 563–578.

[44] 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* (Salvador, Brazil) *(VaMoS '16)*. ACM, New York, NY, USA, 73–80. https://doi.org/10.1145/2866614.2866625

[45] 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* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 48–51. https://doi.org/10.1145/3236405.3237201

[46] Linda M Northrop. 2002. SEI's software product line tenets. *IEEE software* 19, 4 (2002), 32–40.

[47] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Štuikys. 2012. Change Impact Analysis of Feature Models. In *Information and Software Technologies*, Tomas Skersys, Rimantas Butleris, and Rita Butkiene (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 108–122.

[48] 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* (Pisa, Italy) *(VaMoS '13)*. Association for Computing Machinery, New York, NY, USA, Article Article 17, 8 pages. https://doi.org/10.1145/2430502.2430526

[49] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL - A Delta-Based Modeling Framework for Software Product Line Engineering. In *30th International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 852–857. https://doi.org/10.1109/ASE.2015.106

[50] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal foundations for analyzing and refactoring delta-oriented model-based software product lines. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 30:1–30:11. https://doi.org/10.1145/3336294.3336299

[51] Gordon D Plotkin. 2004. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming* 60 (2004), 3–15.

[52] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

[53] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[54] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2019. Partially safe evolution of software product lines. *Journal of Systems and Software* 155 (2019), 17–42.

[55] 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*. 1–5.

[56] 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, Salvador, Brazil, January 27 - 29, 2016*, Ina Schaefer, Vander Alves, and Eduardo Santana de Almeida (Eds.). ACM, 9–16. https://doi.org/10.1145/2866614.2866616

[57] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: tool support for collaborative filtered model-driven software product line engineering. In *Proceedings of the 31st International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 822–827. https://doi.org/10.1145/2970276.2970288

[58] Felix Schwägerl and Bernhard Westfechtel. 2019. Integrated revision and variation control for evolving model-driven software product lines. *Software and Systems Modeling* 18, 6 (2019), 3373–3420. https://doi.org/10.1007/s10270-019-00722-3

[59] Christoph Seidl and Uwe Aßmann. 2013. Towards modeling and analyzing variability in evolving software ecosystems. In *The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013*, Stefania Gnesi, Philippe Collet, and Klaus Schmid (Eds.). ACM, 3:1–3:8. https://doi.org/10.1145/2430502.2430507

[60] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. 2012. Co-evolution of models and feature mapping in software product lines. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*, Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides (Eds.). ACM, 76–85. https://doi.org/10.1145/2362536.2362550

[61] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Capturing Variability in Space and Time with Hyper Feature Models. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems* (Sophia Antipolis, France) *(VaMoS '14)*. Association for Computing Machinery, New York, NY, USA, Article Article 6, 8 pages. https://doi.org/10.1145/2556624.2556625

[62] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore - A Model-Based Delta Language Generation Framework. In *Modellierung 2014, 19.-21. März 2014, Wien, Österreich (LNI)*, Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer (Eds.), Vol. P-225. GI, 81–96. https://dl.gi.de/20.500.12116/17067

[63] 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 - Volume 1* (Florence, Italy) *(SPLC '14)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/2648511.2648514

[64] Herman Steyn. 2002. Project management applications of the theory of constraints beyond critical chain scheduling. *International Journal of Project Management* 20, 1 (2002), 75–80.

[65] Y. Xue, Z. Xing, and S. Jarzabek. 2010. Understanding Feature Evolution in a Family of Product Variants. In *2010 17th Working Conference on Reverse Engineering*. 109–118. https://doi.org/10.1109/WCRE.2010.20