Incremental Feature Model Synthesis for Clone-and-Own Software Systems in MATLAB/Simulink

Alexander Schlie
a.schlie@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Alexander Knüppel
a.knueppel@tu-bs.de
TU Braunschweig
Braunschweig, Germany

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

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

ABSTRACT

Families of related MATLAB/Simulink systems commonly emerge ad hoc using *clone-and-own* practices. Extractively migrating systems towards a *software product line (SPL)* can be a remedy. A *feature model (FM)* represents all potential configurations of an SPL, ideally, in non-technical domain terms. However, yielding a sensible FM from automated synthesis remains a major challenge due to domain knowledge being a prerequisite for *features* to be adequate abstractions. In *incremental* reverse engineering, subsequent generation of FMs may further overwrite changes and design decisions made during previous manual FM refinement.

In this paper, we propose an approach to largely automate the synthesis of a suitable FM from a set of cloned MATLAB/Simulink models as part of reverse engineering an SPL. We fully automate the extraction of an initial, i.e., a *technical*, FM that closely aligns with realization artifacts and their variability, and further provide operations to manually refine it to incorporate domain knowledge. Most importantly, we provide concepts to capture such operations and to replay them on a structurally different technical FM stemming from a subsequent reverse engineering increment that included further systems of the portfolio. We further provide an implementation and demonstrate the feasibility of our approach using two MATLAB/Simulink data sets from the automotive domain.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software reverse engineering**; *Maintaining software.*

KEYWORDS

MATLAB/Simulink, clone-and-own, variability, 150% model, feature model, synthesis, incremental, individual, refinement, mapping

ACM Reference Format:
Alexander Schlie, Alexander Knüppel, Christoph Seidl, and Ina Schaefer. 2020. Incremental Feature Model Synthesis for Clone-and-Own Software Systems in MATLAB/Simulink. 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/10.1145/3382025.3414973

1 INTRODUCTION

In industrial domains, such as avionics, rail and automotive, MATLAB/Simulink models are pivotal for the development of embedded software systems [17, 60]. In these domains, variability is an emergent property, which is oftentimes implemented by copying and subsequently modifying existing systems [3, 18]. Denoted *clone-and-own* [54], families of related models emerge ad-hoc with proper documentation typically not cherished [21, 56, 57]. As a result, information on common and varying parts of models, i.e., *their variability information*, is lost and entailed problems often become evident only in the long run, e.g., increased maintenance overhead [21]. A remedy can be the adoption of a *software product line (SPL)* [49], which reuses commonalities in realization artifacts and models configuration options for variable parts explicitly in a variability model [4, 40, 61]. *Feature models (FMs)* [36] are the most popular notation for variability models, specifying *features* as abstractions of realization artifacts relevant to stakeholders, organized in a tree to depict their relations and dependencies [4, 15, 40, 74]. However, creating an FM for a set of related system variants is a tedious and challenging endeavor [1, 47], especially for large product portfolios.

In domains with many software products stemming from clone-and-own, extractively adopting an SPL, i.e., capitalizing on existing products by extracting common and varying elements, is most prevalent [38, 39]. Due to the sheer size of a product portfolio, extracting *all* products *at once* is typically infeasible. Contrary, the focus may initially be on extracting mission-critical or high-payoff products, whereas remaining systems are incrementally included in the SPL as needed [38]. For such an SPL, an FM can, in principle, be synthesized from existing system variants [6, 25, 72] but it commonly aligns closely with the technical realization of products and does not provide a suitable level of abstraction. To adequately incorporate domain knowledge, such FM must be refined via manual edit operations to abstract from implementation specifics, which is an inherently complex and time-consuming task.

Especially in incremental reverse engineering scenarios, manual design decisions must be preserved for subsequent increments, such that they are not overwritten and established mappings to realization artifacts are not lost. Otherwise, the FM must be recreated repeatedly, which would render the SPL adoption infeasible for families of large and complex systems, e.g., MATLAB/Simulink models. Although work exists to identify variability in MATLAB/Simulink models, the focus is mostly on realization artifacts [2, 35, 53, 60, 67].

To the best of our knowledge, no work exists that copes with an extractive SPL migration of MATLAB/Simulink model variants to facilitate an incremental enrichment of an FM while preserving individual design decisions and maintaining a mapping between realization artifacts and respective features abstracting from them.

We aim to bridge this gap and make the following contributions:

- We propose a novel approach that records edit operations on FMs, and reapplies them upon enlarging the SPL to preserve and mimic prior design decisions. In each reverse engineering increment, we create a *technical feature model (TFM)*, i.e., a problem space representation of realization artifacts and their identified variability. Such TFM is then edited to yield a more sensible FM for the analyzed variants, which we denote the *domain feature model (DFM)*. In each reverse engineering increment, edits are recorded and reapplied in the next increment to converge the newly created TFM towards the constructed DFM. Subsequently, remaining edits supplement existing ones for the next increment.
- We demonstrate the feasibility of our approach using two data sets from the automotive domain. We show our approach to converge created TFMs towards the DFM when incrementally including further variants in the SPL.

The remainder of this paper is structured as follows. We provide background information on MATLAB/Simulink, variability models and prior work relevant to our contribution in Sec. 2. We introduce our approach that records and reapplies edits to converge a TFM towards a DFM in Sec. 3. We detail our feasibility study from the automotive domain and state our research questions in Sec. 4. We discuss produced results in Sec. 5 and state related work in Sec. 6. We detail future work and conclude our paper in Sec. 7.

2 PRELIMINARIES

This section provides background on MATLAB/Simulink, problem and solution space variability models and relevant prior work.

2.1 MATLAB/Simulink Software Systems

MATLAB/Simulink is a block-based behavioral modeling language using ***functional blocks*** and ***signals*** to specify software system functionality. In various industrial domains (e.g., rail, avionic and automotive), such models are pivotal to the development of embedded systems, and are further used to generate source code, simulations and test cases [43, 53, 77]. Each *functional block* of a *MATLAB/Simulink* represents a specific *function* (e.g., *Multiply*) that receives and relays data via incoming and outgoing signals. Every block has a ***label***, i.e., textual name and a non-volatile identifier, the ***simulink identifier (SID)***, which allows for every block to be uniquely identified in the respective model. Moreover, *subsystem (SM)* blocks encapsulate logically connected blocks. SMs can be nested, hence creating a model hierarchy [60], with each block residing on a hierarchical layer. Industrial MATLAB/Simulink models can grow to enormous size and complexity, comprising several thousand blocks [17, 43, 53] with then *hierarchical layers* and more [33].

We depict in Fig. 1a two simple *MATLAB/Simulink* models $M1$ and $M2$, each comprising one SM labeled *Subsystem* and further show the nesting of blocks within SMs. Fig. 1a also schematically illustrates *variability* in $M1$ and $M2$ by circling *varying* blocks in red.

2.2 Solution and Problem Space Variability

For software systems evolved from clone-and own, variability can be represented on different levels of abstraction [16, 50]. Variability within *realization artifacts*, such as source code or functional blocks of MATLAB/Simulink models, facilitates the *solution space* [13]. Abstracting from the solution space, and hence, from realization artifacts, the *problem space* represents variability on a conceptual level, e.g., to facilitate discussions among non-technical stakeholders [16]. For both such spaces, variability models are a major constituent [16].

To represent common and variability-containing parts in the solution space (here MATLAB/Simulink models), *family models* are used, which are also commonly referred to as *150% models* [35]. 150% models are a valuable asset for developing and managing SPLs [49], and are used in de-facto standards like *pure::variants* [51]. We show in Fig. 1b the 150% model for models $M1$ and $M2$ with realization artifacts (here functional blocks) and their variability information annotated. Moreover, Fig. 1b shows varying blocks in $M1$ and $M2$ to be *alternative*, parts contained in both models to be *mandatory* and those only present in one model to be *optional*.

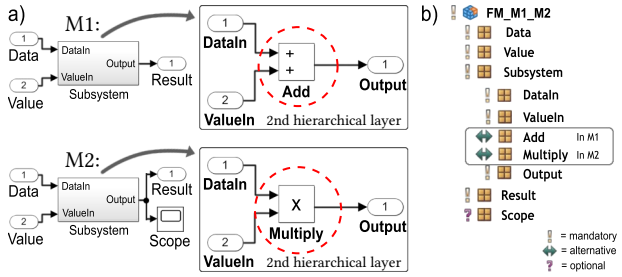


Fig. 1. (a) MATLAB/Simulink Models & (b) their 150% Model

To represent variability in the problem space, various notations exists [15, 52, 69], e.g., *decisions modeling* [14, 22], *orthogonal variability modeling* [32] and the *Common Variability Language* [34]. However, *feature models* prevail as the most prominent notion for managing variability in SPLs [1, 4, 19, 40]. A *feature* is a domain abstraction relevant to stakeholders that reflects characteristic or user-visible behavior of a software system [5, 74]. An FM organizes *features* hierarchically in a *tree*, and further defines their valid combinations, denoted *configurations* [24, 36, 74]. Although 150% models and FMs are both trees, the former reflect variability in the implementation, while the latter represents domain concepts. For a unified variability management in *software product line engineering (SPLE)*, a mapping between the problem and solution space is a prerequisite [8, 62]. The 150% model as shown in Fig. 1 establishes a *1-to-n mapping* with any one 150% model element referencing at least one (i.e. $n > 1$) realization artifact (here functional blocks). Moreover, *one* specific artifact is referenced by exactly *one* 150% model element. For MATLAB/Simulink, we use the block's unambiguous identifier, i.e., its SID, to establish references. Thereby, problem space features are linked to solution space elements (and variations respectively) in the 150% model, which themselves, map to realization artifacts.

2.3 Prior Work on Solution Space Variability

Prior work on re-engineering variability in our group [55, 76, 78–81] focuses on the solution space for MATLAB/Simulink models [64–67]. Prior work encompasses a coarse-grain variability analysis of an entire model portfolio [64], followed by a fine-grained analysis of individual systems [66, 67] and, finally, the combination of both analyses to produce a holistic 150% model that represents solution-space variability for multiple MATLAB/Simulink models [65]. Compiled details on previous work can be found online [11].

The, process to re-engineer solution space variability, i.e., a 150% model from cloned MATLAB/Simulink models is as follows.

First, a coarse-grained analysis aims to reduce the complexity of large-scale models by abstracting from them salient system information to describe such models in a simpler form. Such form can be compared efficiently and allows to capture similar SM structures within the models, while further detecting redundancies, and overall, identifying SM parts that warrant a fine-grained analysis.

Secondly, for parts deemed similar, individual blocks are compared in detail using a fine-grained comparison metric assessing various block properties (e.g., their *label* and *function*) to determine the blocks specific *relation*. By that, variability information of model parts and, hence, of functional blocks is captured at fine-grain.

Finally, similar SM structures within the models and identified redundancies are combined with detailed variability information of respective SM elements and aggregated together to create one holistic 150% model. Such model then represents variability on the solution space for an entire portfolio of MATLAB/Simulink models.

3 FEATURE MODEL SYNTHESIS

In this section, we detail our approach to create a DFM for MATLAB/Simulink variants and to reduce design effort by converging towards it in subsequent reverse engineering increments. By that, we combine an extractive and reactive SPL adoption, by initially extracting a subset of variants and including further variants reactively. We depict the overall workflow of our approach in Fig. 2 and refer to our supplementaries [10] for further material and details.

In a nutshell, our approach transforms, in each increment, a 150% model into a TFM (cf. ① & ② in Fig. 2), which represents all elements of the 150% model and their variability in an FM. In other words, the TFM can be perceived as an FM that displays solution space variability. Such TFM can then be refined (i.e., *edited*) by a domain-engineer to construct from it a DFM, which is an FM that meets the domain engineer's expectation, while all respective edits are recorded (cf. ③, ④ & ⑤ in Fig. 2). Upon the next reverse engineering increment, previously recorded edits are reapplied on the newly created TFM, resulting in an intermediate TFM that converges towards the previously constructed DFM (cf. ⑥ & ⑦ in Fig. 2). Upon finalizing such intermediate TFM, applied edits are further recorded to supplement existing ones (cf. ⑧ in Fig. 2). The process then repeats when incrementally extracting further model variants, hence enlarging the SPL (cf. ⑨ in Fig. 2).

3.1 Transforming the 150% Model into a TFM

We transform a 150% model (cf. Fig. 1) that reflects a certain set of MATLAB/Simulink model variants and their variability information into a solution space variability model, and precisely, a TFM. Thereby, in any reverse engineering increment, the TFM is upon creation, a mere reflection of variability identified in the solution space.

In Tab. 1, we provide properties required to transform a 150% model into a TFM and detail the respective procedure in Alg. 1. We note that line references given in this section are directed to Alg. 1.

Analyses from prior work (cf. Sec. 2.3) create for a set of models K, a 150% model, denoted R_K. We denote TFM_{R_K} to be the the TFM created by Alg. 1 for R_K. Moreover, we denote χ to be a set of model variants not yet considered during a reverse engineering increment.

Tab. 1. Properties to Transform a 150% Model into a TFM

K	Set of MATLAB/Simulink model variants to be migrated in a reverse engineering increment.
χ	Set of MATLAB/Simulink model variants that have not yet been considered in any such increment.
R_K	The *150% model* model derived from prior variability analysis, presenting the set of models variants K.
TFM_{R_K}	Technical feature model reflecting the 150% model R_K.

Both an FM and a 150% model as visualized in Fig. 1 are *trees*. Consequently, we transform a 150% model into a TFM by processing every element of the 150% model separately, beginning with the *root*, and creating for every element a feature in the TFM. Precisely, we denote an element of the 150% model r to be an ordered tuple as stated in Eq. 1, comprising a label l, a variability v (cf. Sec. 2.3), a parent element r_p, child elements r_C (when nesting blocks), references to realization artifacts *Refs* (i.e., functional blocks), alternative properties Alt_p (e.g., varying labels) and alternative elements Alt_r, if present.

$$r : \langle l, v, Alt_p, Alt_r, Refs, r_p, r_C \rangle \qquad r \in R \tag{1}$$

Similarly, we denote a feature f to be an ordered tuple as stated in Eq. 2, which comprises the same information of r, such as a parent feature f_p and child features f_C, but in addition, exhibits a classification c (whether f is *abstract* or *concrete*) and a description d.

$$f : \langle l, v, c, d, Alt_f, Refs, f_p, f_C \rangle \qquad f \in FM \tag{2}$$

To retrieve any element from a tuple (which can be a set), we utilize the notation given in Eq. 3, which shows the retrieval of the set of referenced realization artifacts *Refs* from a 150% model element r.

$$x \leftarrow r(Refs) \qquad \texttt{x is new set} \tag{3}$$

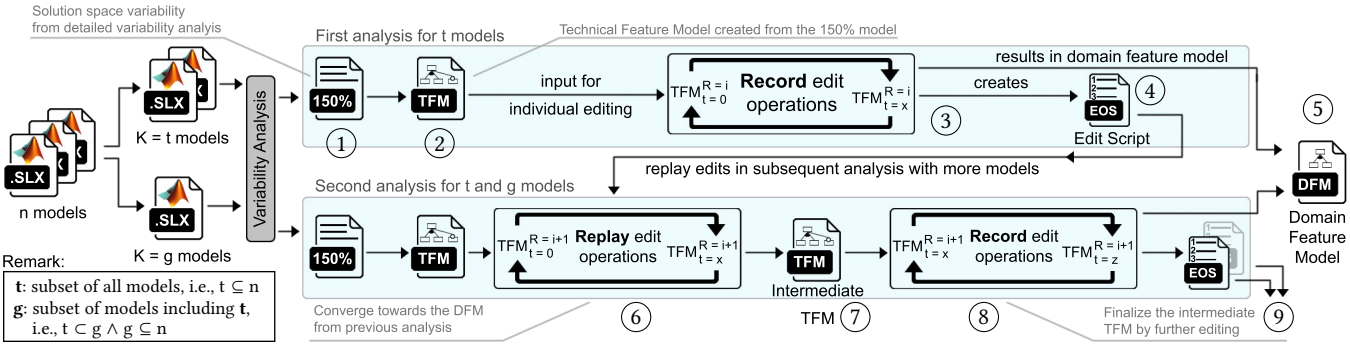


Fig. 2. Schematic Workflow to Record and Replay Edits to Converge to a Domain-Engineered Feature Model

As depicted in line 3 of Alg. 1, we begin with the respective root element r_{root} of the 150% model R and create a feature f_{root} for it. The root f_{root} is assigned the label (i.e. $f_{root}(l)$) of the 150% model, which is a concatenation of names from all model variants comprised in the 150% model. Overall, Alg. 1 processes the 150% model R in a recursive and top down fashion (cf. lines 6 & 25). Beginning with the root element of the 150% model, its child elements $r_{root}(r_C)$ are retrieved in line 6. For each element r, we create a new feature f_{new} in line 8 and, for instance, assign it the label of r, its variability v and transfer references to realization artifacts $r(Refs)$ held by r. Thereby, upon creation, a feature in the TFM holds references to realization artifacts (here functional blocks). When encountering alternatives elements in the 150%, hence elements r constituting a *variation point* (cf. Fig. 1), we retrieve the associated identifier λ of such *variation point* in line 10.

If for such variation point, a feature f_{VP} has already been created (cf. lines 15 to 17), we assign to it f_{new}, created for the current *alternative* element r, as a child feature f_C in line 12. Hence, *alternative* elements r are grouped as child features f_C under their corresponding *variation point* feature f_{VP}. For each created feature, we set its parent-child relations in line 21 according to the parent child-relation of the respective element r. Therefore, the parent-child relations are the same between elements r of the 150% model R and features f in the derived TFM. Finally, if the current element r nests further blocks (cf. line 24), the process repeats recursively.

We show in Fig. 3 the TFM derived by Alg. 1 for the 150% model shown in Fig. 1b. The TFM in Fig. 3 comprises all elements of the 150% model from Fig. 1b and, hence, references to all realization artifacts of the respective MATLAB/Simulink models shown in Fig. 1a. Corresponding to the 150% model R, no two *features* in the TFM can exist that reference the *same* element $r \in R$, i.e., realization artifact.

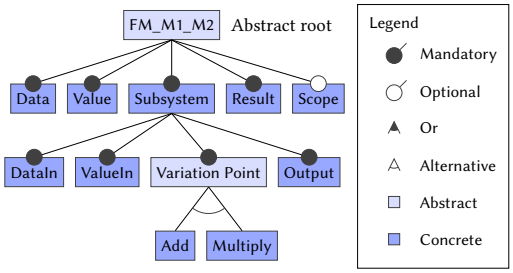


Fig. 3. TFM for the 150% Model from Fig. 1

Moreover, the TFM in Fig. 3 retains all parent-child relations and shows variations according to Fig. 1. In Fig. 3, features labeled *Add* and *Multiply* each reference exactly one functional block from their corresponding model, whereas the feature labeled *Subsystem*, references two realization artifacts. Precisely, the *Subsystem* feature, being present in both models (cf. Fig. 1b) holds two distinct references to each individual *Subsystem* block from both models in Fig. 1a. Moreover, Fig. 3 depicts optional features (here labeled *Scope*) for the respective optional element in the 150% model from Fig. 1b.

3.2 Edit Operations to Refine the TFM

To successively transform the TFM into a suitable DFM, the respective TFM needs to be edited to adequately contain domain knowledge. We provide eleven edit operations, i.e., *refinements* and

Input: R_K - 150% model reflecting the set of models variants K
Output: TFM_{R_K} - TFM representing the 150% model R_K

1 $(V, E) \leftarrow \emptyset$
2 $TFM_{R_K} \leftarrow (V, E)$
3 $r_{root} \leftarrow getRootNode(R_T)$
4 $f_{root} \leftarrow \langle r_{root}(l), mandatory, abstract, \emptyset, \emptyset, K, \emptyset, \emptyset \rangle$
5 $V \leftarrow V \cup f_{root}$
6 Process child elements($\beta \leftarrow r_{root}(C_r)$) :
7 **foreach** $r \in \beta$ **do**
8 $f_{parent} \leftarrow \emptyset$
9 $f_{new} \leftarrow \langle r(l), r(v), concrete, \emptyset, r(Alt_p), r(Refs), \emptyset, \emptyset \rangle$
10 **if** $r(v) = alternative$ **then**
11 $\lambda \leftarrow r(Alt_{id})$
12 **if** $\exists f \in TFM_{R_K}(V) : f(l) = \lambda$ **then**
13 $f_{parent} \leftarrow f$
14 **end**
15 **else**
16 $f_{VP} \leftarrow \langle \lambda, alternative, abstract, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$
17 $f_{VP}(v) \leftarrow mandatory$
18 $f_{parent} \leftarrow f_{VP}$
19 **end**
20 **end**
21 **else**
22 $f_{parent} \leftarrow getFeatureFor(r(r_p))$
23 **end**
24 $f_{parent}(f_C) \leftarrow f_{parent}(f_C) \cup f_{new}$
25 **if** $|r(C_r)| > 0$ **then**
26 Process child elements($\beta \leftarrow r(C_r)$) :
27 **end**
28 **end**
29 **return** TFM_{R_K}

Algorithm 1: Transforming a Family Model into a TFM

one auxiliary operation, that is the selection / deselection of features, which all can be applied to an FM to alter its structure (e.g., by relocating features) or to modify single features (e.g., by *renaming* them). Overall, our edit operations coincide with those provided by sophisticated feature modeling tools such as FeatureIDE [41], whereas customized operations are out of scope. We list properties required for edit operations in Tab. 2, denoting them *edits* hereafter.

Every such edit q is associated with an individual *validity function* $\mathcal{V}$, which evaluates to *true* if q can be applied given the current state of the FM. Therefore, one can perceive $\mathcal{V}$ as a set of rules that must be fulfilled. For instance, one could not rename any feature f to an empty string, as this would violate $\mathcal{V}$ defined for such edit. For the FM editing, we aim to be non-restrictive, also when combing edits. Although out of scope, we acknowledge orthogonal work on anomaly and consistency checking, e.g., with FeatureIDE [41].

Tab. 2. Entities Required for Edit Operations

q	An atomic edit operation to modify an FM.
Q	The set of atomic edit operations q.
f_{tar}	A feature, on which the user performs an atomic edit q.
f_{root}	The current root feature of the FM.
F_{sel}	The features f, which are currently selected by the user.
$\mathcal{V}$	A validity function that asserts whether q can be applied.

Overall, edits q can be applied to an FM in an arbitrary order and, they can further be *undone*, for instance to revise design decisions at a later point in time. Edits q must not be configuration-preserving, as we do not focus the refactoring of an FM, but rather its evolution, which is inherently driven by the users' individual design decisions. The individual edit operations q we provide are as follows.

Add feature: Adds a new feature f_{new} to the FM either (a) *above* or (b) *below* a selected feature f_{tar}. Upon creation, f_{new} holds no references to realization artifacts, is assigned an automatically generated label l and has no description d. By default, the variability $f_{new}(v)$ is set to *optional*, unless f_{new} is the new *root*, in which case it is *mandatory*). The respective validity function $\mathcal{V}$ is given in Eq. 4.

$$\mathcal{V} = \left\{ \begin{array}{ll} true & f_{tar} \in FM \\ false & otherwise \end{array} \right\} \quad (4)$$

Remove feature: Removes a feature f_{tar} from the FM. Upon deletion of f_{tar}, held references to realization artifacts are removed as well, if f_{tar} holds such references. However, we note that removing references in the FM neither removes elements from the 150% model nor realization artifacts themselves. All features except the *root* can be removed and the validity function $\mathcal{V}$ is given in Eq. 5.

$$\mathcal{V} = \left\{ \begin{array}{ll} true & f_{tar} \in FM \wedge f_{tar} \neq f_{root} \\ false & otherwise \end{array} \right\} \quad (5)$$

Remove feature trunk: Recursively applies the *remove feature* edit on the entire *subtree* of f_{tar}, while not removing f_{tar} itself. Consequently, the validity function $\mathcal{V}$ stated in Eq. 5 is applied.

Fuse features: For selected features F_{sel}, their references to realization artifacts are transferred, i.e., *fused*, into the target feature f_{tar}. Subsequently, features F_{sel} are removed from the FM. To cope with complex MATLAB/Simulink models, in which functionality is typically implemented on multiple hierarchical layers, we allow for features to be fused, also if they are no direct children of f_{tar}. We show in Fig. 4a features *DataIn*, *ValueIn* and *Output* selected, and in Fig. 4b, for them to be *fused* into the feature *Subsystem*.

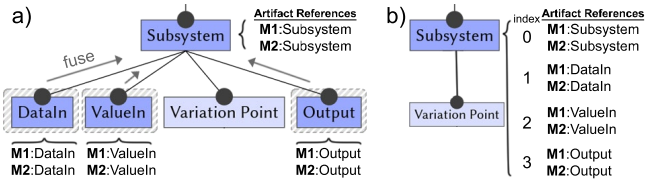


Fig. 4. Fusing Features and Transferring Artifact References

Shown in Fig. 4, referenced realization artifacts are ordered in a *stack*, with references existing within f_{tar} being located at the top of such stack, and also to remain there after *fusing*. Consequently, the first (i.e., top) element in such a stack always references the realization artifacts used upon creation of f_{tar}. Moreover, all further referenced realization artifacts can be unequivocally stated to be the result of *fusing*. The validity function to fuse features is given in Eq. 6.

$$\mathcal{V} = \left\{ \begin{array}{ll} true & f_{tar} \in FM \wedge |F_{sel}| > 1 \wedge \forall f_x \in F_{sel} : f_x \in FM \\ & \wedge f_{tar} \notin F_{sel} \\ false & otherwise \end{array} \right\} \quad (6)$$

Split feature: For every referenced realization artifact of the feature f_{tar}, a new feature f_{new} is created, which is added as a child feature under the parent of f_{tar}. A feature f_{tar} can only be split if it holds at least two artifact references, i.e., it has been fused before and, hence, the validity function applies as given in Eq. 7 applies.

$$\mathcal{V} = \left\{ \begin{array}{ll} true & f_{tar} \in FM \ \wedge \ |f_{tar}(Refs)| \geq 2 \\ false & otherwise \end{array} \right\} \quad (7)$$

Holding four artifact references, Fig. 5a shows the feature *Subsystem*, which in Fig. 5b, has been split, here yielding three separate features.

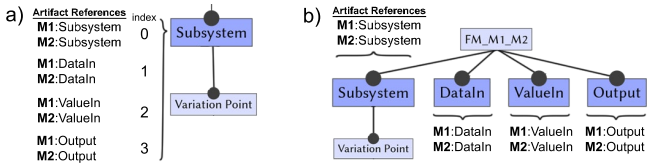


Fig. 5. Splitting Features and Artifact References

Edit variability: The variability of f_{tar} can be set to *mandatory* or *alternative* and the validity function $\mathcal{V}$ applies as stated in Eq. 5.

Edit variability group: The variability group of the feature f_{tar} is set be either an *or*, *and* or *alternative* group. The validity function $\mathcal{V}$ applies as stated in Eq. 4.

Edit classification: Any feature f_{tar}, except the root feature f_{root}, can be set to be either *abstract* or *concrete*. Therefore, the validity function $\mathcal{V}$ applies as given in Eq. 5.

Edit visibility: For a feature f_{tar}, its direct children f_C are hidden, or made visible respectively. Once hidden, a feature can not be modified directly, e.g., by renaming it, but indirectly, e.g., when it is part of a branch that is *removed* entirely from the FM. Overall, the validity function $\mathcal{V}$ applies as given in Eq. 8.

$$\mathcal{V} = \left\{ \begin{array}{ll} true & f_{tar} \in FM \wedge f_{tar} \ is \ visible \\ false & otherwise \end{array} \right\} \quad (8)$$

Move feature: Selected features F_{sel} are relocated, and hence, *moved* under the target feature f_{tar}. We allow for any feature to be moved except f_{root}, however, we do not allow to move a parent feature, such that it would become a child of itself. Overall, the validity function $\mathcal{V}$ to move features is given in Eq. 9.

$$\mathcal{V} = \left\{ \begin{array}{ll} true & f_{tar} \in FM \wedge f_{tar} \notin F_{sel} \wedge f_{tar} \neq f_{root} \\ & \wedge \forall f_{sel} \in F_{sel} : f_{sel} \notin f_{tar}(f_C) \\ false & otherwise \end{array} \right\} \quad (9)$$

Upon moving any f_{sel}, its children, i.e., $f_{sel}(f_C)$ are also moved, such that any feature subtree is not separated by our *move* operation.

Edit label/description: Resets the label or description of the feature f_{tar} to a string specified by the user. For convenience, labels must not be unique when editing the FM, while features can be distinguished by references to realization artifacts. Upon exporting the FM for usage in other tools, it is post-processed to assure unique labels. Overall, the validity function V applies as stated in Eq. 4.

3.3 Record Edits While Building the DFM

During individual refinement of the FM, i.e., editing the TFM to construct a DFM for a certain set of model variants, we record all edits applied by the user within an ordered sequence. By that, we preserve domain knowledge induced to create the DFM and, hence, the user's individual design decisions. Moreover, by recording respective edit operations, we further provide means for individual edits within sequences to be reverted, revised, replaced or,

in general, to be reevaluated. We refer to a sequence of recorded edits as an *edit operation sequence (EOS)*. We provide properties required to construct EOSs while deriving a DFM from a TFM in Tab. 3, complementing properties given in Tab. 2 and Tab. 1.

Tab. 3. Properties for Editing the TFM to Build the DFM

$TFM_{R_K}^{t=i}$	TFM, initially generated from the 150% model R_K, after a total of i edit operations have been applied.
δ : $\langle v_1, v_2 \rangle$	A tuple containing for a certain property of a feature (i.e., it's label) the value prior to editing such property v_1 and it's new value v_2.
Δ : $\langle f, q, \delta, t, FM \rangle$	A recorded edit of an FM, being a tuple comprising a feature f in a specific FM, on which at time t, an edit q is applied with an optional value tuple δ.
Ω	The EOS and, hence, an ordered set of tuples Δ recorded during the modification of the TFM_{R_K}.
Φ_{R_K}	Ordered set of EOSs Ω that have been applied in the process of transforming $TFM_{R_K}^{t=0}$ to DFM_{R_K}.
DFM_{R_K}	Refined TFM_{R_K}, reflecting the domain engineer's expectation of a final FM for the set of systems K.
ϕ	An ordered set of all EOSs Ω created.

We further present the respective procedure in Alg. 2, to which all given line references within this section are directed. Upon editing any FM in general (here domain-engineering a DFM from a TFM), the user needs to manually initiate and terminate the recording, while EOSs are automatically generated in the background. For large data sets, constructing the DFM in one session might be infeasible. Thus, EOSs can be chained, which allows to reason about intermediate results and to discuss them, prior to further editing.

Shown in Alg. 2, we first set the time t in line 3, so that when working in sessions, and hence, EOSs already exist (cf. line 2), the time stamp for each edit operation is set correctly. Subsequently, we create a set Ω in line 5 to hold all future edit operations. Upon user-individual editing and, hence, domain-engineering of the TFM in line 6 (in principle any FM), every applied edit q is recorded and a corresponding tuple Δ is created in line 7 to further store the time t the respective feature f changed properties δ, if present, and the current state of the TFM. Simultaneously, the edit Δ is stored within the set of edits Ω in line 8 and the feature model changes respectively in line 9. When finished editing, all previously induced edits, and thereby design decisions, constitute for the TFM to become the DFM in line 13. Finally, the set ϕ of all EOSs Ω applied to transform the initial TFM (at time $t = 0$) into the DFM is created and subsequently stored in the set of all EOSs Φ in lines 14 and 15. We show in Fig. 6 the DFM, which is domain-engineered by editing the TFM from Fig. 3 and further depict in Fig. 6 the respective EOS and, hence, individual edits used to *refine* the TFM from Fig. 3.

3.4 Replay Edits to Converge Towards the DFM

Upon enlarging, the SPL and, hence, including further variants in a subsequent reverse engineering increment, we automatically create a new TFM (cf. Alg. 1) for the enlarged model set $K \cup \chi$ (cf. Tab. 1). To replay previously recorded EOSs on such new TFM, features in respective change operations Δ must be identified within the new TFM stemming from the subsequent increment. Thereby, any

Input: TFM_{R_K} [1] - See legend in algorithm below for description
Output: DFM_{R_K} - Domain-engineered FM for the 150% model R_K

1 $t \leftarrow 0$
2 **foreach** $\phi \in \Phi$ **do**
3 $\quad t \leftarrow t + |\phi|$
4 **end**
5 $\Omega \leftarrow \emptyset$
6 **while** *applying edit operations* $q \in Q$ **do**
7 $\quad \Delta \leftarrow \langle f, q, \delta, t, TFM_{R_j}^{t} \rangle$
8 $\quad \Omega \leftarrow \Omega \cup \Delta$
9 $\quad TFM_{R_K}^{t+1} \leftarrow \epsilon(\Delta, TFM_{R_K}^{t})$
10 $\quad t \leftarrow t + 1$
11 **end**
12 $DFM_{R_K} \leftarrow TFM_{R_K}^{t}$
13 $\phi \leftarrow (TFM_{R_K}, DFM_{R_K}, \Omega)$
14 $\Phi \leftarrow \Phi \cup \phi$
15 **return** DFM_{R_K}

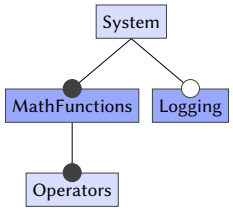
1 TFM presenenting the 150% model R_K, which was generated for the set of input models K (cf. Alg. 1)

Algorithm 2: Creating a DFM by Editing the TFM

Δ and its comprised edit q can be replayed if and only if a mapping can be established between the feature $f_{original}$ referenced in Δ and any one feature f present in the new TFM. For this, features labels do not suffice, as they may change, and so can any feature's location, parent-child relation and variability. Hence, we identify features by exploiting the references to realization artifacts *Refs* held by each feature. We consider this viable as no two features can exist in an FM, which reference the same realization artifact (i.e., functional block). We show the corresponding procedure to replay EOSs in Alg. 3, which aims to converge towards the DFM. We refer to our online material [10] for a detailed depiction of a second reverse engineering increment for our example in Figs. 3 & 6.

As depicted in Alg. 3, we process all EOSs ϕ in Φ and, for each EOS ϕ, try to replay contained change operations Δ. To locate the appropriate feature to replay Δ on within $TFM_{R_{K\cup\chi}}$, we retrieve the original feature $f_{original}$ from Δ and the set of referenced realization artifacts $Refs_{original}$ it holds. We evaluate whether, within $TFM_{R_{K\cup\chi}}$ a feature f exists, for which its set of referenced realization artifacts $f(Refs)$ is equal to $Refs_{original}$ and, hence, a feature that references the *same* functional blocks. If such feature f exists, we use the validity function $\mathcal{V}$ to check whether the respective edit q can be applied on f. If $\mathcal{V}$ evaluates to true, we use the replay function ϵ to mimic user input to change the $TFM_{R_{K\cup\chi}}$ and its feature f according to the edit q. If the validity function $\mathcal{V}$ fails, we prompt the user to inspect the conflict and to resolve it manually.

To address such conflicts during replaying, users can replace, insert or revise individual conflicting change operations, along



t	feature f	q	δ
1	FM_M1_M2	*Rename*	v_1: FM_M1_M2 v_2: **System**
2	Variation Point	*Rename*	v_1: Variation Point v_2: **Operators**
3	Operators	*Hide* f_C	
4,5,6	DataIn, ValueIn, Output	*Select*	
7	Subsystem	*Fuse* f_{sel}	
⋮	Remaining edits and a 2nd increment are online [10]		

Fig. 6. DFM based on the TFM in Fig. 3 and Excerpt from EOS

with the possibility to disregard and proceed. If no feature f can be found in the $TFM_{R_{K\cup\chi}}$ that comprises the *same* set of referenced artifacts, but rather a subset only, e.g., after features are split (cf. Sec. 3.2), we apply validity function $\mathcal{V}$ accordingly and prompt the user to decide whether to replay Δ. If the user decides no, EOSs can be refined, e.g., by *removing* or *replacing* single edits. After replaying all EOSs, an intermediate TFM is derived that, with every replayed change operation, converged towards the DFM. Such intermediate TFM is then subject to Alg. 2 and further editing by the user to finalize a DFM for the enlarged model set $K \cup \chi$.

Input: $TFM_{R_{K\cup\chi}}$[1], Φ[2] - See legend in algorithm below for description
Output: $DFM_{R_{K\cup\chi}}$ - DFM for enlarged set of input models $\{K \cup \chi\}$

1 **foreach** $\phi \in \Phi$ **do**
2 **foreach** $\Delta \in \Omega$ **do**
3 $\langle f, q, \delta, t, TFM^{t}_{R_K} \rangle \leftarrow \Delta$
4 $f_{original} \leftarrow \Delta(f)$
5 $Refs_{original} \leftarrow f_{original}(Refs)$
6 **if** $\exists f \in TFM^{\Delta(t)}_{R_{K\cup\chi}} : Refs_{original} \equiv f(Refs)$ **then**
7 **if** $\mathcal{V}(f, \Delta(q))$ **then**
8 $TFM^{\Delta(t)+1}_{R_{K\cup\chi}} \leftarrow \epsilon(\Delta, TFM^{\Delta(t)}_{R_{K\cup\chi}})$
9 **end**
10 **else**
11 Goto line 23
12 **end**
13 **end**
14 **else if** $\exists f \in TFM^{\Delta(t)}_{R_{K\cup\chi}} : Refs_{original} \subset f(Refs)$ **then**
15 **if** $\mathcal{V}(f, \Delta(q))$ **then**
16 Give notice for user to decide
17 **if** user approves **then**
18 $TFM^{\Delta(t)+1}_{R_{K\cup\chi}} \leftarrow \epsilon(\Delta, TFM^{\Delta(t)}_{R_{K\cup\chi}})$
19 **end**
20 **end**
21 **else**
22 manually resolve
23 **end**
24 **end**
25 **end**
26 **end**
27 **return** $DFM_{R_{K\cup\chi}}$

[1] TFM for the family model R, which represents an enlarged set of input models with $\{K \cup \chi\}$
[2] Edit operations from the previous iteration for smaller set of input models K, resulting in DFM_{R_K}

Derive by using the intermediate $TFM^{\Delta(t)+1}_{R_{K\cup\chi}}$ as input for Alg. 2, that is for final editing

Algorithm 3: Replaying Edit Operations on a TFM

4 EVALUATION

In this section, we state our objectives and provide information on the analyzed models and used data analysis guidelines [59].

4.1 Research Questions

With our approach, we record edit operations applied on the TFM during its transformation towards a DFM and replay such edit operations upon repeating the process for an enlarged set of model variants. With our evaluation, we do not assess performance as *user-specific* editing would account for the majority of runtime. Moreover, we do not evaluate the reduction of modeling effort as, for this, a comprehensive user-study with domain-engineers is a prerequisite. Overall, we focus on the following research questions.

RQ1: *To what extent can recorded edit operation sequences (EOSs) be replayed on a TFM created for an enlarged set of model variants?*
The ability to replay change operations Δ is pivotal for our approach to be feasible. We refer to the *extent* as the number of Δs that can be replayed in relation to all Δs recorded in EOSs.

RQ2: *To what extent do replayed EOSs converge the TFM created for an enlarged set of model variants towards the existing DFM?*
Replayed EOSs must converge the new TFM towards the DFM, rather than diverge from it. We denote the *extent* to be the number of features in the DFM, which can be mapped to features in the intermediate TFM and, hence, after replaying.

RQ3: *What is the impact on the existing configuration space upon finalizing the DFM for the enlarged set of model variants?*
For our approach to be feasible, existing configurations must not be rendered void by replaying EOSs. With this question, we seek to evaluate whether existing configurations can be preserved.

4.2 Setup

To evaluate the feasibility of our approach and its ability to converge towards a DFM, we conducted a case study with two data sets of MATLAB/Simulink models from the automotive domain. We use an exemplary *driver assistance system (DAS)* from the publicly available SPES_XT [46] project and an *auto platooning system (APS)* from the CrESt [45] project. For both data sets, we artificially generated model variants by identifying self-contained parts within the models and extracting them. We list the parts used for variant composition in Tab. 4 and state the number of blocks #*blocks* and subsystems #*SMs* comprised, the hierarchical layer H_D the respective part resides on and the number of hierarchical layers it nests #nH_D. Using the project documentation of both data sets, we identified dependencies between parts as stated in Tab. 4, e.g., between parts *DIS* and *FTS*. Respecting such dependencies, we created a total of 19 variants for the DAS model and nine variants for the APS model.

Tab. 4. DAS & APS Model Parts used for Variant Composition

Model name & *Abbreviation*	#blocks	#SMs	H_D	#nH_D
BrakeAssist *'BA'*	9	1	1	2
DistanceWarner *'DW'*	15	1	1	2
Distronic *'DIS'* (*req.* TM, FTS)	70	4	1	2
EmergencyBrake *'EB'*	32	1	1	2
FollowToStop *'FTS'*	19	2	1	3
Limiter *'LIM'*	114	15	1	5
Tempomat *'TM'* (*req.* FTS)	293	32	1	6
VelocityControl *'VC'*	29	0	1	1
↑ **DAS** ↑ 625 blocks, max. H_D: 7		↓ **APS** ↓ 254 blocks, max. H_D: 4		
FuelConsumption *'FC'*	17	0	3	0
MaxAcceleration *'MA'*	12	0	3	0
DrivingResistance *'DR'*	20	0	3	0
DesiredAcceleration *'DA'*	16	0	3	0
Adap.CruiseControl *'ACC'* (*req.* CACC)	28	0	2 & 3	0
ExtendedCuiseControl *'CACC'*	17	1	2	1

SMs – subsystem blocks **H_D** – hierarchical layer **#nH_D** – # of nested H_Ds ***req.*** - requires

For instance, the largest variants created for both data sets comprise all respective parts listed in Tab. 4, whereas other variants contain only one part, e.g., *FTS*, or two parts, e.g., *FC & MA*. We note that the largest APS model variant comprises 254 blocks while the largest DAS model variant contains a total of 625 blocks. Further specifics on the size of model variants can be found online [10]. We evaluated our approach using a Dual-Core™i7 processor with 12 GB of RAM and Windows™7 on 64bit.

4.3 Data Analysis Guidelines

For each data set separately, we initially synthesize a TFM for two model variants and refine, i.e., *edit*, it towards a DFM and record every edit operation within an EOS. For the APS data set, we subsequently enlarge the set of model variants by one, hence, considering a model variant that was not included in prior increments and synthesize a TFM respectively. For the DAS data set, we proceed analogously but include *two* model variants in each increment. For this, an exponential number of combinations is possible, for which a manual assessment is infeasible. For the DAS model, we started with the two largest variants and, for each increment, randomly selected *two* remaining variants. Contrary, for the APS model, we began with the smallest variants. Our approach allows for arbitrary increment strategies, and we do not dictate any order. Hence, we focus in our evaluation on the two possible extremes, that is beginning with either the smallest model variants or with the largest respectively. Consequently, this yields 10 increments for the APS data set and eight for the DAS data set.

In each increment step, we re-apply all previously recorded EOSs on the generated TFM. By that, we can assess, for each increment, the extent to which edit operations can be replayed (RQ1) and further compare the resulting intermediate TFM with the DFM (RQ2). Upon finalizing the intermediate TFM, i.e., constructing the enlarged DFM, we export it to FeatureIDE [41] to calculate configurations, which, for each increment, can be compared with prior ones (RQ3). Results were assessed and evaluated by an expert well familiar with the DAS and APS models. We embedded our approach in our toolkit [63], which is based on Java [73], Eclipse [29], its modeling framework [30] and rich client platform [31].

5 RESULTS AND DISCUSSION

In this section, we state and discuss produced results. We show aggregated data and provide information and screencasts online [10].

RQ1: Quantity of Replaying Edit Operations. Our approach facilitates on the recording of edit operations applied during a prior analysis increment and to replay such edits subsequently when further model variants are included. In Tab. 5, we show the analysis increments $\mathcal{I}$ for both the DAS and APS data sets and state the size of generated TFMs for respective increments. Moreover, Tab. 5 lists the number of edit operations applied in each increment, denoted *local* edits, which combined with all previously recorded *local* edits, facilitate the *total* number of edits. Therefore, in each increment $\mathcal{I}$, the corresponding *local* edits reflect one specific EOS created to refine respective TFM (cf. Fig. 6). For instance, Tab. 5 shows that for the first two analyzed DAS model variants $v0$ and $v1$ ($\mathcal{I} = 1$), the generated TFM comprised 432 features. To refine such TFM, a total of 531 edit operations were applied, yielding the DFM for the analyzed model variants. Here, no prior edits exist and, hence, none can be replayed. We note that it is plausible for the number of edit operations to exceed that of features, as edits are captured at fine-grain level, such as selecting and deselecting features. Tab. 5 shows that, for the DAS data set, EOSs were created only for the first two increments, comprising four model variants $v0 - v3$. Furthermore, Tab. 5 depicts that, for the second increment $\mathcal{I}$=2, all 531 edit operations from the previous increment could be replayed. To finalize the resulting intermediate TFM (cf. Fig. 2), an additional 87 edits were required, yielding two EOSs with a total of 618 edit operations. In subsequent increments, no further edits were required, but the two EOSs were sufficient to converge any TFM created for an enlarged variant set towards the previously constructed DFM. Hence, no further EOSs were created. For the APS data set, the first analysis increment $\mathcal{I}$ comprised the two smallest variants, for which the generated TFM contained 256 features. Contrary to the DAS data set, each increment $\mathcal{I}$ and, thus, every enlargement of the model variant set, required further edits. For instance, the final increment for the APS data set $\mathcal{I} = 8$ comprising all APS model variants and required an additional three edit operations to converge towards the DFM. In such increment, one edit operation from the seven previously recorded EOSs, totaling 338 edits, could not be replayed on the TFM for variants $v0 - v8$. Precisely, the generated 150% model render one realization artifact void and, consequently, in the generated TFM, on which previously recorded edits for the respective feature reflecting such artifact could not be replayed.

Tab. 5. Size of TFMs and EOSs for Data Sets

$\mathcal{I}$	TFMs variants v	TFMs # features	# edits local	# edits total	# replayable edits
1	v0 - v1	432	531	531	-
2	v0 - v3	503	87	618	531 (100%)
3	v0 - v5	503	-	618	618 (100%)
⋮	Results for subsequent TFMs and EOSs are identical to that of $\mathcal{I} = 3$				
9	as the inclusion of variants, i.e., $v6 - v18$, requires no further editing				
	↑ **DAS** ↑		↓ **APS** ↓		
1	v0 - v1	256	280	280	-
2	v0 - v2	270	18	298	280 (100%)
3	v0 - v3	272	7	305	298 (100%)
4	v0 - v4	226	14	319	305(100%)
5	v0 - v5	211	8	327	319 (100%)
6	v0 - v6	232	6	333	327 (100%)
7	v0 - v7	250	5	338	333 (100%)
8	v0 - v8	252	3	341	**337** (99.7%)

Overall, the results in Tab. 5 show that the vast majority of the EOSs (predominantly comprising the *fusing*, *renaming*, *changing variability*, *moving* and the auxiliary *selection* of features) could be replayed on newly created TFMs to converge them towards the DFM.

RQ2: Converging Towards the DFM by Replaying Edits. In addition to the quantity of replayable edit operations, it is of great importance that their replaying results in the intermediate TFM to converge towards the previously created DFM. In Tab. 6, we provide respective information and detail the intermediate TFMs created by replaying all EOSs recorded during previous increments $\mathcal{I}$.

Upon finalizing such intermediate TFM in each increment, the thereby constructed DFMs are further listed in Tab. 6. Moreover, we denote C in Tab. 6 to reflect the extent to which an intermediate TFM converges towards the DFM created during the previous increment. In other terms, C states to what extent the current DFM (i.e., *feature tree*) from the increment $\mathcal{I}$ is comprised within the intermediate TFM from the subsequent increment $\mathcal{I} + 1$ (i.e., also a feature tree), which considered further model variants. Moreover, we denote $\mathcal{S}$ to state the overall similarity of the DFM from the increment $\mathcal{I}$ and the intermediate TFM from increment $\mathcal{I} + 1$, by combing C with the overall difference in size of both model FMs.

Precisely, $\mathcal{S}$ is calculated by dividing the number of features of the DFM with the number of features of the intermediate TFM and multiplying that with C. In addition, in Tab. 6, we list the number of identified errors $\mathcal{E}$ in the intermediate TFM from the respective increment. For instance, Tab. 6 shows that for the first increment for both data sets, no intermediate TFM exists, as no EOS exists. For the first increment $\mathcal{I} = 1$ of the DAS data set, the DFM constructed for model variants $v0$ and $v1$ comprises 15 features in total. For the second increment, all 531 edits recorded during the first increment (cf. Tab. 5) were reapplied on the generated TFM for the four model variants $v0$ to $v3$. Depicted in Tab. 6, such edits reduce the size of the original TFM for increment $\mathcal{I} = 2$ from 503 (cf. Tab. 5) to 84 features in the resulting intermediate TFM.

Moreover, Tab. 6 shows that the DFM was contained within the intermediate TFM, albeit one error $\mathcal{E}$ was identified upon inspection of the intermediate TFM. Correspondingly, Tab. 6 depicts results for the APS data set and shows intermediate TFM to always comprise the DFM (i.e., $C = 100\%$) and to converge towards (i.e., $\mathcal{S}$ increases). Tab. 6 only lists the number of errors we observed in the intermediate TFMs and, upon closer inspection, we identified our solution-space variability analysis to be the cause, rather than our approach to record and replay edits. Respective errors were mostly limited to different variability classifications of individual features (i.e., *optional* or *mandatory*). Overall, we conservatively consider the consecutive replaying of recorded EOSs to be acceptable as our results show them to successfully converge TFMs towards a DFM.

Tab. 6. Intermediate TFMs and DFMs for Data Sets

$\mathcal{I}$	Intermediate TFMs variants v	# features	DFM variants v	# features	C	$\mathcal{S}$	$\mathcal{E}$
1	-	-	v0 - v1	15	-	-	-
2	v0 - v1	84	v0 - v3	15	100%	18%	1
3	v0 - v3	15	v0 - v5	15	100%	100%	0
⋮	Results for subsequent intermediate TFMs are identical to that of $\mathcal{I} = 3$						
9	as two EOSs suffice to fully converge towards the DFM (cf. Tab. 5)						
	↑ DAS ↑		↓ APS ↓				
1	-	-	v0 - v1	9	-	-	-
2	v0 - v1	23	v0 - v2	10	100%	39%	0
3	v0 - v2	14	v0 - v3	13	100%	71%	1
4	v0 - v3	13	v0 - v4	10	100%	77%	2
5	v0 - v4	11	v0 - v5	10	100%	91%	1
6	v0 - v5	12	v0 - v6	11	100%	83%	2
7	v0 - v6	12	v0 - v7	11	100%	92%	2
8	v0 - v7	11	v0 - v8	11	100%	100%	2

RQ3: Configuration Preserving. For our approach to be feasible, replaying previously recorded edit operations should not render existing configurations void. In Tab. 7, we provide the number of configurations for DFMs and intermediate TFM for both the DAS and APS data sets. For instance, Tab. 7 shows the first DFM created for models $v0$ and $v1$ of the DAS data to exhibit 32 configurations. Moreover, the intermediate TFM for $v0 - v3$ derived by reapplying the EOS recorded during previous DFM construction exhibits only eight configurations. The decrease in configurations corresponds to the error $\mathcal{E}$ observed within the intermediate TFM (cf. Tab. 6) and, precisely, it results from an *optional* feature to be falsely classified as *mandatory*. However, Tab. 6 shows that for the DAS data set, the second intermediate TFM created for variants $v0 - v3$ exhibits no errors (cf. Tab. 6) and fully converges towards the DFM, thereby preserving *all* configurations. For the APS data set shown in Tab. 7, the number of configurations for the DFM increased upon enlarging the analyzed model set. Precisely, the first DFM for APS model variants $v0$ and $v1$ exhibits four features while, upon finalizing it when including $v2$ ($\mathcal{I} = 2$), it exhibits eight configurations (cf. Tab. 6 to note increasing number of features respectively). Moreover, Tab. 7 shows that configurations for intermediate TFMs of the DAS data set vary, which corresponds to the errors $\mathcal{E}$ observed therein (cf. Tab. 6). Similar to our observations for the DAS data set, a false variability classification of a single feature within the intermediate TFM $v0 - v8$ for the APS data set resulted in configurations to be lost.

Tab. 7. Number of Configurations for DAS & APS Data Sets

$\mathcal{I}$	variants	# configuration	
		Intermediate TFM	DFM
1	v0 - v1	-	32
2	v0 - v3	8	32
3	v0 - v5	32	32
⋮	Results for subsequent intermediate TFMs are identical to that of $\mathcal{I} = 3$		
9	as two EOSs suffice to fully converge towards the DFM (cf. Tab. 5)		
		↑ DAS ↑	↓ APS ↓
1	v0 - v1	-	4
2	v0 - v2	4104	8
3	v0 - v3	32	8
4	v0 - v4	64	8
5	v0 - v5	16	8
6	v0 - v6	16	16
7	v0 - v7	16	16
8	v0 - v8	6	16

However, we argue that such errors could, in each increment $\mathcal{I}$, be resolved with fewer edits (cf. Tab. 5). Overall, we observe that our approach can preserve existing configurations and, in instances where such are rendered void (e.g., for $v0 - v8$ of the APS data set), such problems can be addressed quickly (cf. three edits in Tab. 5).

5.1 Threats to Validity

Following guidelines in [59], threats to validity are inherently present for our approach, and we discuss them in paragraphs below.

Internal Validity: We provide edit operations, and we acknowledge that users may judge differently on their completeness and soundness. To mitigate threats regarding completeness, we adopted operations provided by FeatureIDE [41], a sophisticated modeling tool. Nevertheless, we acknowledge that further operations may exist that we did not consider. Regarding the soundness of our edit operations, we argue that they must not be configuration-preserving, as we address the evolution of an FM by means of individual design decisions, rather that the refactoring of an FM. Hence, certain edit operations can render existing configurations void, if respective edits are applied by the user. However, our evaluation has shown that for the analyzed data sets, configurations can be preserved.

Construct Validity: To evaluate our approach, we used specific orders, in which we included variants in the analysis. While we acknowledge that different orders may impact the results, our prior work mitigates this threat by constructing the 150% model, used as basis for our approach independently of input order [64, 65].

External Validity: For our evaluation, we used synthetic variants from one domain only. We acknowledge that other domains

and models may exhibit peculiarities we did not consider that limit the generalizability of our approach. We argue that the variants exhibit a considerable complexity and are, at least to some extent, representative for models from the automotive domain. Moreover, the modifications we performed on an FM must not reflect practices employed by others and respective operations can vary in number and scope for every user. However, we argue that our approach specifically aims to provide such freedom, recognizing feature modeling as an inherently creative process.

Conclusion Validity: For the evaluated data sets, we consider our approach to be feasible, but acknowledge that experts may judge differently on its quality and practical usage. We argue that we have shown our approach to cope with different increment strategies, in both size and order, to mitigate, at least to some extent, threats regarding quality. To mitigate threats regarding the practical usage of our approach, we argue that evaluated models exhibit a considerable size and complexity and that results provide at least an indication on our approaches' behavior in an industrial setting. However, we acknowledge that a comprehensive study with domain experts is needed to answer questions about qualitative aspects, the usefulness of results and how domain information included in the DFM could be useful for the evolution of the SPL.

6 RELATED WORK

In addition to prior work on solution-space variability (cf. Sec. 2.3), we acknowledge feature-location techniques [20, 58]. However, our approach differs from respective work as we allow for features to be defined by users via modeling an FM, initially retrieved from variability inherent to products' implementations. Similar to our approach, Martinez et al. [39] describes the location and extraction of features from a family of models. However, their technique is fragile against changes as they assume no modifications to features (e.g., refactorings) between variants. In difference, the editing we provide specifically enables the incorporation of such domain knowledge to cope with respective modifications. Furthermore, Font et al. [25] stress that many automatic variability extraction approaches concerned with model variants fail to adequately reflect the expectations of domain experts. To reduce manual effort imposed on domain experts, Font et al. apply heuristic techniques from operations research and search-based software engineering (e.g., genetic algorithms) to further converge towards a realistic representation [26–28]. We overcome the inaccuracy of such approaches by proposing a semi-automatic and incremental technique that allows domain experts to manually edit an initially extracted TFM.

Most importantly, we try to eliminate redundant manual work by recording atomic edit operations (i.e., the applied domain knowledge) systematically and replay them on subsequent TFMs created for further reverse engineering increments. To the best of our knowledge, such incremental approaches capturing, i.e., *preserving*, replayable domain knowledge (i.e. *design decisions*) at such fine gain to reduce redundant labor are scarce in the scientific community. Finally, we benefit from a closely related field regarding the restructuring of FMs, namely product-line evolution, where a new version of an FM is derived by applying a sequence of edit operations similar to our approach. Bürdek et al. [12] identify a total of 19 atomic edit operations for the feature-model dialect supported by FeatureIDE [42]. Kehrer et al. [37] present an approach to automatically derive a set of consistency-preserving edit operations for a given modeling language including FMs. These works do not explicitly cover the co-evolution of FMs and references to their realization artifacts, such that operations as *fusing* and *splitting* features are not addressed directly. However, some taxonomies of edit operations for FM evolution acknowledge the importance of such operations [9, 68, 82] and Seidl et al. [71] provide a selection of such operations, which partially inspired our work. Apart from that, many reasoning techniques [23, 48, 75] on FM edits to ensure consistency throughout the manual editing are also applicable in our context as-is.

Metzger et al. [44] formalize the relationship of orthogonal variability models (OVMs) to FMs to facilitate an automated reasoning, e.g., to assert Boolean satisfiability. In difference, our work focuses on the incremental inclusion of model variants into an FM, for which work by Metzger et al. can be complementary. Similar to our approach, Bécan et al. [7] aim to synthesize FMs, albeit based on ontologies. In contrast to our work, the FM is synthesized and refactored once, whereas we focus on evolving the FM over various reverse engineering increments, by preserving design decisions. Schroeter et al. [70] promote a *multi-perspective* approach for FMs that is representing it differently for varying scenarios, e.g., technical vs. non-technical. In difference, our approach focuses on the evolution of an FM and its changes over reverse engineering increments.

7 CONCLUSION AND FUTURE WORK

FMs are a constituent part of an SPL, but yielding a sensible FM from an automated synthesis of multiple related software system variants remains a major challenge. Moreover, during an incremental reverse engineering, subsequently synthesized FMs may overwrite previously induced design decisions and, hence, domain knowledge.

We address this problem and, focusing on MATLAB/Simulink models, largely automate the synthesis of a suitable FM from a set of related model variants as part of reverse engineering an SPL. We automatically derive a *technical* FM (TFM) that reflects realization artifacts and their variability. Furthermore, users can individually *edit* such TFM to incorporate domain knowledge to construct a *domain* FM (DFM). Most importantly, we capture all such edits and replay them on a structurally different TFM stemming from a subsequent analysis that included further model variants. We demonstrate the feasibility of our approach using two MATLAB/Simulink data sets from the automotive domain and show our approach to capture all edits applied during reverse engineering increments and to, in each such increment, to converge the newly created TFMs towards the DFM by reapplying previously recorded edits.

In future work, we plan to incorporate *constraints* by reasoning about occurrences and dependencies of realization artifacts associated with features. Moreover, we aim to create configurations and to subsequently derive new MATLAB/Simulink model variants by exploiting the selected feature's references to realization artifacts. Finally, we aim to incorporate domain experts in our approach to assess its usefulness and overall applicability in industrial settings.

ACKNOWLEDGMENTS

This work was supported by the DFG(German Research Foundation) (SCHA 1635/12-1). We thank Oliver Urbaniak and Kamil Rosiak.

REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. 2010. Composing Feature Models. In *Proc. of the Intl. Conference on Software Language Engineering (SLE)*. Springer, Berlin, Heidelberg, 62–81.

[2] M. Alalfi, E. Rapos, A. Stevenson, M. Stephan, T. Dean, and J. Cordy. 2014. Semi-automatic Identification and Representation of Subsystem Variability in Simulink Models. In *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 486–490.

[3] M. H. Alalfi, J. R. Cordy, and T. R. Dean. 2014. Analysis and Clustering of Model Clones: An Automotive Industrial Experience. In *Proc. of the Intl. Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 375–378.

[4] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. 2012. Efficient synthesis of feature models. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. 106–115.

[5] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin Heidelberg.

[6] W. Assunção, R. Lopez-Herrejon, L. Linsbauer, S. Vergilio, and A. Egyed. 2017. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering* 22 (02 2017).

[7] G. Bécan, M. Acher, B. Baudry, and S. Ben Nasr. 2015. Breathing Ontological Knowledge Into Feature Model Synthesis: An Empirical Study. *Empirical Software Engineering* 21 (03 2015).

[8] K. Berg, J. Bishop, and D. Muthig. 2005. Tracing Software Product Line Variability: From Problem to Solution Space. In *Proc. of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT)*. South African Institute for Computer Scientists and Information Technologists, 182–191.

[9] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski. 2010. EvoFM: Feature-Driven Planning of Product-Line Evolution. In *Proc. of the Int. Workshop on Product Line Approaches in Software Engineering (PLASE)*. 24–31.

[10] A. Schlie - TU Braunschweig. 2020. Supplementary Material Website. https://www.isf.cs.tu-bs.de/cms/team/schlie/material/splc2020.

[11] A. Schlie - Technical University Braunschweig. 2020. Prior Work Supplementary Material Website. https://www.isf.cs.tu-bs.de/cms/team/schlie/material.

[12] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr. 2016. Reasoning About Product-Line Evolution using Complex Feature Model Differences. *ASEJournal* 23, 4 (2016), 687–733.

[13] L. Cleophas, D.G. Kourie, V. Pieterse, I. Schaefer, and B.W. Watson. 2016. Correctness-by-Construction Taxonomies Deep Comprehension of Algorithm Families. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*. Springer, 766–783.

[14] Software Productivity Consortium Services Corporation. 1993. *Reuse-driven Software Processes Guidebook: SPC-92019-CMC, Version 02.00.03*. Software Productivity Consortium Services Corporation.

[15] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proc. of the Intl. Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 173–182.

[16] Y. Dajsuren and M. van den Brand. 2019. *Automotive Systems and Software Engineering: State of the Art and Future Trends*. Springer.

[17] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. 2010. Model Clone Detection in Practice. In *Proc. of the Intl. Workshop on Software Clones (IWSC)*. 57–64.

[18] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. 2008. Clone Detection in Automotive Model-Based Development. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 603–612.

[19] D. Dhungana and P. Grünbacher. 2008. Understanding Decision-Oriented Variability Modelling. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. 233–242.

[20] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. 2013. Feature Location in Source Code: a Taxonomy and Survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.

[21] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.

[22] S. El-Sharkawy, S. Dederichs, and K. Schmid. 2012. From Feature Models to Decision Models and Back Again an Analysis Based on Formal Transformations. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 126–135.

[23] G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. 2002. Consistency-Preserving Model Evolution through Transformations. In *International Conference on the Unified Modeling Language*. Springer, 212–227.

[24] W. Fenske, T. Thüm, and G. Saake. 2014. A Taxonomy of Software Product Line Reengineering. In *Proc. of the Intl. Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. 1–8.

[25] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. 2015. Building Software Product Lines from Conceptualized Model Patterns. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 46–55.

[26] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. 2016. Feature Location in Model-Based Software Product Lines through a Genetic Algorithm. In *International Conference on Software Reuse*. Springer, 39–54.

[27] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. 2016. Feature Location in Models through a Genetic Algorithm Driven by Information Retrieval Techniques. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 272–282.

[28] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. 2017. Achieving Feature Location in Families of Models through the use of Search-Based Software Engineering. *IEEE Transactions on Evolutionary Computation* 22, 3 (2017), 363–377.

[29] Eclipse Foundation®. 2020. Eclipse Foundation Website. https://eclipse.org.

[30] Eclipse Foundation®. 2020. Eclipse Modelling Framework Website. https://eclipse.org/modeling/emf.

[31] Eclipse Foundation®. 2020. Eclipse Rich Client Platform Website. https://wiki.eclipse.org/Rich_Client_Platform.

[32] V. Gruhn and R. Striemer. 2018. *The Essence of Software Engineering*. Springer International Publishing.

[33] A. Haber, C. Kolassa, P. Manhart, P.M.S. Nazari, B. Rumpe, and I. Schaefer. 2013. First-class Variability Modeling in Matlab/Simulink. In *Proc. of the Intl. Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 11–18.

[34] Ø. Haugen, A. Wąsowski, and K. Czarnecki. 2013. CVL: Common Variability Language. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 277.

[35] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. 2014. Family Model Mining for Function Block Diagrams in Automation Software. In *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*. ACM, 36–43.

[36] Kyo K., Sholom C., James H., William N., and A. P. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[37] T. Kehrer, G. Taentzer, M. Rindt, and U. Kelter. 2016. Automatically Deriving the Specification of Model Editing Operations from Meta-Models. In *Proc. of the Int. Conference on Theory and Practice of Model Transformations (TPMT)*. Springer, 173–188.

[38] C.W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *Software Product-Family Engineering*. Springer, Berlin, Heidelberg, 282–293.

[39] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. l. Traon. 2015. Automating the Extraction of Model-Based Software Product Lines from Model Variants. In *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. 396–406.

[40] R. Mazo, C. Salinesi, D. Diaz, and A. Lora Michiels. 2011. Transforming Attribute and Clone-Enabled Feature Models Into Constraint Programs Over Finite Domains. In *Proc. of the Intl. Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. 188–199.

[41] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer International Publishing.

[42] J. Meinicke, T. Thüm, R. Schröter, S. Krieter, F. Benduhn, G. Saake, and T. Leich. 2016. FeatureIDE: Taming the Preprocessor Wilderness. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 629–632.

[43] D. Merschen, A. Polzer, G. Botterweck, and S. Kowalewski. 2011. Experiences of Applying Model-based Analysis to Support the Development of Automotive Software Product Lines. In *Proc. of the Intl. Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 141–150.

[44] A. Metzger, K. Pohl, P. Heymans, P. Schobbens, and G. Saval. 2007. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Proc. of the Int. Requirements Engineering Conference (RE)*. IEEE, 243–253.

[45] Technical University Munich. 2020. Collaborative Embedded Systems. https://crest.in.tum.de/.

[46] Technical University Munich. 2020. Software Platform Embedded Systems 'XT'. http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html.

[47] D. Nešić, J. Krüger, S. Stănciulescu, and T. Berger. 2019. Principles of Feature Modeling. In *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 62–73.

[48] M. Nieke, C. Seidl, and T. 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*. 48–51.

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

[50] R. Pohl, M. Höchsmann, P. Wohlgemuth, and C. Tischer. 2018. Variant Management Solution for Large Scale Software Product Lines. In *Proc. of the Int. Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 85–94.

[51] Pure-Systems®. 2020. Pure::Variants Website. http://www.pure-systems.com.

[52] R. Rabiser. 2019. Feature Modeling vs. Decision Modeling: History, Comparison and Perspectives. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 134–136.

[53] R. Reicherdt and S. Glesner. 2012. Slicing MATLAB Simulink Models. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 551–561.

[54] C. Riva and C. D. Rosso. 2003. Experiences with Software Product Family Evolution. In *Proc. of the Joint Workshop on Software Evolution and Intl. Workshop on Principles of Software Evolution (IWPSE-EVOL)*. IEEE, 161–169.

[55] K. Rosiak, O. Urbaniak, A. Schlie, C. Seidl, and I. Schaefer. 2019. Analyzing Variability in 25 Years of Industrial Legacy Software: An Experience Report. In *Proc. of the Int. Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)*. ACM, 65–72.

[56] J. Rubin and M. Chechik. 2012. Combining Related Products into Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE) (LNCS, Vol. 7212)*. Springer, 285–300.

[57] J. Rubin and M. Chechik. 2013. Quality of Merge-Refactorings for Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE) (LNCS, Vol. 7793)*. Springer, 83–98.

[58] J. Rubin and M. Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*. Springer, 29–58.

[59] P. Runeson and M. Höst. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164.

[60] U. Ryssel, J. Ploennigs, and K. Kabitzsch. 2010. Automatic Variation-point Identification in Function-block-based Models. In *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 23–32.

[61] Marco S. and Sybren D. 2007. Classifying Variability Modeling Techniques. *Information and Software Technology (IST)* 49, 7 (2007), 717–739.

[62] F. Sanen, E. Truyen, and W. Joosen. 2009. Mapping Problem-Space to Solution-Space Features: A Feature Interaction Approach. In *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 167–176.

[63] A. Schlie, K. Rosiak, O. Urbaniak, I. Schaefer, and B. Vogel-Heuser. 2019. Analyzing Variability in Automation Software with the Variability Analysis Toolkit. In *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*. ACM, 191–198.

[64] A. Schlie, S. Schulze, and I. Schaefer. 2018. Comparing Multiple MATLAB/Simulink Models Using Static Connectivity Matrix Analysis. In *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 185–196.

[65] A. Schlie, C. Seidl, and I. Schaefer. 2019. Reengineering Variants of MATLAB/Simulink Software Systems. In *Security and Quality in Cyber-Physical Systems Engineering*. Springer, 267–301.

[66] A. Schlie, D. Wille, L. Cleophas, and I. Schaefer. 2017. Clustering Variation Points in MATLAB/Simulink Models Using Reverse Signal Propagation Analysis. In *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Springer, 77–94.

[67] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer. 2017. Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 215–224.

[68] K. Schmid and H. Eichelberger. 2008. A Requirements-based Taxonomy of Software Product Line Evolution. *Electronic Communications of the EASST* 8 (2008).

[69] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (2007), 456–479.

[70] J. Schroeter, M. Lochau, and T. Winkelmann. 2012. Multi-Perspectives on Feature Models. In *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer-Verlag, Berlin, Heidelberg, 252–268.

[71] C. Seidl, F. Heidenreich, and U. Aßmann. 2012. Co-evolution of Models and Feature Mapping in Software Product Lines. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 76–85.

[72] S. She, K. Czarnecki, and A. Wąsowski. 2012. Usage Scenarios for Feature Model Synthesis. In *Proc. of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone (VARY)*. ACM, 15–20.

[73] Oracle Systems®. 2020. Java Website. https://www.java.com/en/.

[74] T. Thum, D. Batory, and C. Kastner. 2009. Reasoning about Edits to Feature Models. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 254–264.

[75] T. Thum, D. Batory, and C. Kastner. 2009. Reasoning about Edits to Feature Models. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 254–264.

[76] K. Wehling, D. Wille, C. Seidl, and I. Schaefer. 2018. Reducing Variability of Technically Related Software Systems in Large-Scale IT Landscapes. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. ACM, 224–235.

[77] M.W. Whalen, A. Murugesan, S. Rayadurgam, and M.P.E. Heimdahl. 2014. Structuring Simulink Models for Verification and Reuse. In *Proc. of the Intl. Workshop on Modeling in Software Engineering (MISE)*. ACM, 19–24.

[78] D. Wille, Ö. Babur, L. Cleophas, C. Seidl, M. van den Brand, and I. Schaefer. 2018. Improving Custom-Tailored Variability Mining Using Outlier and Cluster Detection. *Science of Computer Programming* 163 (2018), 62–84.

[79] D. Wille, T. Runge, C. Seidl, and I. Schulze. 2017. Extractive Software Product Line Engineering Using Model-Based Delta Module Generation. In *Proc. of the Intl. Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 36–43.

[80] D. Wille, S. Schulze, C. Seidl, and I. Schaefer. 2016. Custom-Tailored Variability Mining for Block-Based Languages. In *Proc. of the Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 271–282.

[81] D. Wille, K. Wehling, C. Seidl, and I. Schaefer. 2017. Variability Mining of Technical Architectures. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 39–48.

[82] Y. Xue, Z. Xing, and S. Jarzabek. 2010. Understanding Feature Evolution in a Family of Product Variants. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 109–118.