The Journal of Systems and Software 147 (2019) 46–63

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Controversy Corner

Retest test selection for product-line regression testing of variants and versions of variants

Sascha Lity*, Manuel Nieke, Thomas Thüm, Ina Schaefer

TU Braunschweig, Institute of Software Engineering and Automotive Informatics, Braunschweig, Germany

ARTICLE INFO

Article history:
Received 19 April 2018
Revised 7 August 2018
Accepted 28 September 2018
Available online 3 October 2018

Keywords:
Regression testing
Software evolution
Software product lines

ABSTRACT

Testing is a crucial activity of product-line engineering. Due to shared commonality, testing each variant individually results in redundant testing processes. By adopting regression testing strategies, variants are tested incrementally by focusing on the variability between variants to reduce the overall testing effort. However, product lines evolve during their life-cycle to adapt, e.g., to changing requirements. Hence, quality assurance has also to be ensured after product-line evolution by efficiently testing respective versions of variants. In this paper, we propose retest test selection for product-line regression testing of variants and versions of variants. Based on delta-oriented test modeling, we capture the commonality and variability of an evolving product line by means of differences between variants and versions of variants. We exploit those differences to apply change impact analyses, where we reason about changed dependencies to be retested when stepping from a variant or a version of a variant to its subsequent one by selecting test cases for reexecution. We prototypically implemented our approach and evaluated its effectiveness and efficiency by means of two evolving product lines showing positive results.

1. Introduction

Quality assurance, e.g., achieved via testing (Harrold, 2000), is a crucial and challenging activity during *software product line* (SPL) engineering (Pohl et al., 2005). SPL engineering (Pohl et al., 2005) is a paradigm for managed artifact reuse to realize a software family with well-defined commonalities and differences by means of features, i.e., customer-visible functionality (Kang et al., 1990), already applied successfully in industry (Linden et al., 2007; Weiss, 2008). The varying assembly of those artifacts results in a vast number of different, yet similar variants for which testing has to be applied. Due to the inherent commonality shared between variants, testing each variant separately results in redundant testing processes and, thus, in an increasing testing effort (McGregor, 2001).

Recent SPL testing techniques (Oster et al., 2011; Engström, 2010; da Mota Silveira Neto et al., 2011; Lee et al., 2012) reduce the testing effort, e.g., by selecting only a sample of variants to increase the coverage of combinatorial feature interactions (Johansen et al., 2012; Al-Hajjaji et al., 2016a; Rothberg et al., 2016). However, each variant is still tested individually without exploiting the commonality and obtained test artifacts as well as test results between variants. To overcome this drawback, *regression testing strategies* (Yoo and Harman, 2007) are applicable to facilitate incremental SPL testing (Tevanlinna et al., 2004; Engström, 2010; Lochau et al., 2014; Baller et al., 2014; Lity et al., 2016b; Lachmann et al., 2015; Varshosaz et al., 2015; Damiani et al., 2017) and, thus, to exploit the inherent reuse potential also during testing, e.g., by incorporating differences between variants for test-case prioritization (Lachmann et al., 2015), test-case generation (Varshosaz et al., 2015), or model-based SPL testing (Lochau et al., 2014; Damiani et al., 2017). Thus, the resulting reduction of the test effort enables a more targeted use of the limited test resources.

Furthermore, quality assurance also has to be ensured after SPL *evolution*. Software product lines evolve over time to adapt to changing requirements, customer requests, or novel technologies (Svahnberg and Bosch, 1999; Mens et al., 2014). Each evolution step impacts reusable domain artifacts and their interdependencies due to necessary changes. Hence, testing becomes more challenging for SPLs as not a single system is influenced by those changes, but rather the set of variants. To avoid testing the new SPL version completely without incorporating the test artifacts and results of preceding versions, again, regression testing strategies (Yoo and Harman, 2007) are applicable to exploit not only the reuse potential between variants, but also between versions of variants (Runeson and Engström, 2012). Thus, the testing redundancy between SPL versions is tackled to reduce the testing effort when testing variants and versions of variants incrementally.

* Corresponding author.
E-mail addresses: lity@isf.cs.tu-bs.de, lity@ips.cs.tu-bs.de (S. Lity).

https://doi.org/10.1016/j.jss.2018.09.090
0164-1212/© 2018 Elsevier Inc. All rights reserved.

In this paper, we propose retest test selection for model-based SPL regression testing of variants and versions of variants. The combination of model-based testing (Utting and Legeard, 2006) and retest test selection as regression testing strategy (Yoo and Harman, 2007) facilitates (1) the automatic generation of test cases based on test models, i.e., the behavioral specification of a *system under test* (SUT), and (2) the reduction of test cases to be (re-)executed for testing an SUT guided by applying change impact analysis. For test modeling, we apply (higher-order) delta modeling (Clarke et al., 2015; Lity et al., 2016a) capturing the commonality and variability of variants and their evolution by means of change operations called deltas. As common in model-based testing (Utting and Legeard, 2006), a test model is created manually based on the requirements of the SUT. For delta-oriented test models Clarke et al. (2015); Lity et al. (2016a), the creation process can be supported to make the approach more practically applicable to clones and product lines in general, either (1) by applying family mining (Wille et al., 2017) for delta extraction if a set of variant-specific models already exist, (2) by applying model differencing (Pietsch et al., 2015) facilitating the delta extraction during the modeling process, or (3) by transforming between variability implementation techniques (Fenske et al., 2014; Lity et al., 2018) if the SUT is already developed using another implementation technique (Schaefer et al., 2012).

We exploit the explicit knowledge of difference by means of deltas to apply two kinds of automated change impact analyses to reason about the retest of test cases. First, we utilize incremental model slicing (Lity et al., 2015) to automatically identify changed dependencies between variants during incremental testing of *one SPL version under test*. Changed dependencies denote behavior potentially influenced by changes to the test model. Second, we automatically reason about higher-order delta application (Lity et al., 2016a) to examine changes to the set of variants between *subsequent SPL versions under test* by means of removals, additions, and modifications of variants. Removed variants become obsolete for testing and, thus, we focus on added variants denoting new variants as well as on modified variants representing the versions of previous variants to be tested. For new and modified variants, we apply incremental model slicing to identify influenced behavior to be retested in order to reduce the testing redundancy by exploiting the shared commonality between new variants as well as between versions of variants.

To this end, we incorporate delta-oriented test modeling and change impact analyses in a model-based SPL regression testing framework. We exploit the change impact results to automatically select test cases to be retested. The reexecution of test cases validates that new behavior is implemented as specified and that already tested behavior is not influenced other than intended. Our framework allows for incremental testing of one SPL version as well as subsequent SPL versions under test facilitating the reuse of test artifacts and test results to reduce the overall testing effort by tackling the potential redundancy during SPL testing. We prototypically implement and evaluate our approach by means of two evolving SPLs to validate its effectiveness and efficiency in a controlled experiment. In summary, our contributions are as follows:

- A *retest test selection* technique applicable for regression testing of both variants and versions of variants.
- Its integration in a *model-based SPL regression testing framework* using *incremental change impact analyses* to guide the selection process.
- An evaluation of the *effectiveness* and *efficiency* of our framework by means of two evolving delta-oriented SPLs.

The general concept of our framework for testing one SPL version, i.e., incremental testing of variants by applying retest test selection, has already been presented in previous work (Lochau et al., 2014; Lity et al., 2016b). In this paper, we substantially extend this work by adapting the framework to allow for testing one SPL version and subsequent SPL versions incrementally by reusing test artifacts and test results of preceding testing processes. Thus, we enhance the test-modeling formalism, the change impact analysis, and the workflow of our testing framework. We further extend the evaluation as we now investigate *evolving* delta-oriented SPLs. In addition, we extend higher-order delta modeling (Lity et al., 2016a) by supporting application conditions for deltas and, hence, we redefine the automated reasoning about higher-order delta application to take application conditions into account for the reasoning process.

The remainder of the paper is structured as follows. In Section 2, we describe (higher-order) delta modeling used as test-modeling formalism. In Section 3, we explain our incremental change impact analyses. In Section 4, we propose our retest test selection technique and its incorporation for incremental testing of variants and versions of variants. In Section 5, we present our evaluation. In Section 6, we discuss related work. In Section 7, we conclude the paper.

2. Evolving delta-oriented software product lines

In this section, we first describe the basic notion of SPLs. In addition, we explain delta modeling as a technique to capture variability of variants and versions of variants by the same means. We apply delta modeling as test-modeling formalism for our SPL regression testing framework (cf. Section 4) and further exploit its concept to define incremental change impact analyses (cf. Section 3) to guide our retest test selection.

Software product lines: The SPL engineering (Pohl et al., 2005) paradigm is becoming more importance over the last years (Linden et al., 2007; Weiss, 2008). SPLs allow for a large-scale reuse of domain artifacts resulting in a shorter time-to-market and a reduction of development costs. Based on a set of *features* $F = \{f_1, \ldots, f_n\}$, i.e., customer-visible functionality (Kang et al., 1990), a family of similar software systems $\mathbb{V} = \{v_0, \ldots, v_m\}$ is defined sharing common and variable artifacts to be assembled. The set of variants $\mathbb{V}$, also known as problem space (Czarnecki and Eisenecker, 2000) is specified, e.g., by using a feature model *FM* (Kang et al., 1990; Benavides et al., 2010). To this end, each variant $v_i \in \mathbb{V}$ is represented by a specific feature configuration $F_{v_i} \subseteq F$ such that the configuration satisfies the structure and dependencies defined in the feature model *FM*.

Example 1. Consider the feature model *FM* shown in Fig. 1a, where four configurations are derivable, namely $F_{v_{core}} = \{f_0, f_1\}, F_{v_1} = \{f_0, f_1, f_3\}$, $F_{v_2} = \{f_0, f_2\}$, and $F_{v_3} = \{f_0, f_2, f_3\}$, defining the variant set $\mathbb{V} = \{v_{core}, v_1, v_2, v_3\}$.

In contrast to the problem space, the solution space specifies the set of variable domain artifacts and their mappings to specific features (Czarnecki and Eisenecker, 2000). This mapping controls the valid assembling of artifacts w.r.t. a given feature configuration F_{v_i}. For the development of SPLs with variable artifacts, various variability implementation techniques exist (Schaefer et al., 2012). In this paper, we focus on transformational approaches (Schaefer et al., 2012), i.e., delta modeling (Clarke et al., 2015). Such approaches are well-suited (1) for dealing with variability of variants and versions of variants, (2) to facilitate incremental change impact analysis by focusing on the differences, i.e., the variability, which are explicitly specified, and (3) to build the basis for our model-based SPL regression testing framework (cf. Section 4) due to the knowledge about differences between variants and versions of variants to be exploited for retest test selection.

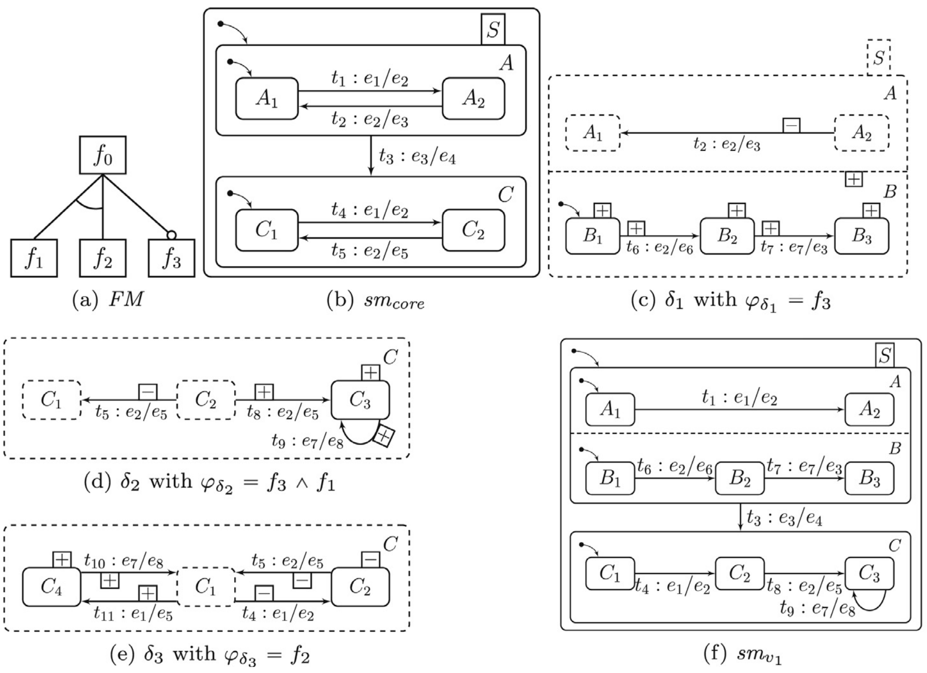


Fig. 1. (a) Feature model *FM* comprising root feature f_0, alternative features f_1 and f_2 and optional feature f_3. (b) Core state machine representing the behavioral specification of variant v_{core}. (c)–(e) Deltas to transform sm_{core}. (f) State machine for variant v_1.

Delta modeling is a flexible and modular modeling formalism, where differences between variants and between versions of variants are explicitly specified in terms of transformations called *deltas* (Clarke et al., 2015; Seidl et al., 2014; Lity et al., 2016a). Due to its artifact-independency, delta modeling is already applied to various types of SPL domain artifacts, e.g., state machines (Lochau et al., 2012), software architectures (Lochau et al., 2014; Lachmann et al., 2015), and performance-annotated activity diagrams (Kowal et al., 2015). In this paper, we focus on its adaptation for state machines as our regression testing technique (cf. Section 4) uses state machines as test-modeling formalism as common in model-based testing (Utting and Legeard, 2006). We first describe delta modeling (Clarke et al., 2015) as technique to model variant-specific state machines in the next paragraph. Afterwards, we describe its extension for modeling SPL evolution.

Modeling variability of variants: In general, the application of delta modeling (Clarke et al., 2015) is based on a predefined *core model*, which, in our case, is a state machine $sm_{core} \in SM$ specifying the behavior, i.e., the reaction on incoming events with corresponding outgoing events, of a *core variant* $v_{core} \in \mathbb{V}$. By *SM*, we refer to the set of all variant-specific state machines. To transform the core sm_{core} into a state machine sm_i of variant $v_i \in \mathbb{V}$, deltas $\delta_j \in \Delta$ are defined, where Δ represents the set of all deltas of an SPL under consideration. A *delta* $\delta = (OP_\delta, \varphi_\delta)$ captures *change operations* $OP_\delta = \{op_1, \dots, op_k\} \subseteq OP$, such as additions ($\mathsf{add}\, e$) or removals ($\mathsf{rem}\, e$) of elements, i.e., states and transitions. For brevity, we abstract from the modification of elements ($\mathsf{mod}(e, e')$) and encode such change operations as a removal and addition of the (modified) element. By *OP*, we refer to the set of all change operations defined over the set of all elements of the current SPL, where further $OP \subset \mathbb{OP}$ holds, i.e., *OP* is a subset of the universe of all possible change operations $\mathbb{OP}$.

In addition, each delta has an *application condition* $\varphi_\delta \in \mathbb{B}(F)$, i.e., a Boolean expression over features, indicating for which feature (configuration) the delta has to be applied. Based on a given configuration F_{v_i}, for each delta $\delta_j \in \Delta$ its application condition is evaluated whether F_{v_i} satisfies φ_{δ_j}. If φ_δ is satisfied, the delta δ_j is selected and gathered in a variant-specific delta set Δ_{v_i}. Afterwards, the delta set Δ_{v_i} is subsequently applied to sm_{core} in a predefined order resulting in the respective variant-specific state machine sm_i. We refer the reader to the literature (Clarke et al., 2015) for a more detailed description and discussion about delta application and conflict handling between change operations to be applied. Based on those definitions, we specify a *delta model* $DM = (sm_{core}, \Delta)$ of an SPL as tuple comprising its core state machine and the set of applicable deltas.

For testing purposes (cf. Section 4), it is further of interest to determine the differences between arbitrary variants, which are derivable in terms of *model regression deltas* Δ_{v_{i-1},v_i} by incorporating their delta sets $\Delta_{v_{i-1}}$ and Δ_{v_i} using an adaption of the symmetric difference. We refer the reader to prior work (Lochau et al., 2012; 2014) for the construction of model regression deltas.

Example 2. Consider the core model shown in Fig. 1b. Three deltas are defined depicted in Fig. 1c to e to realize the variants from Ex. 1, where + represents an addition and − denotes a removal of an element. By applying $\Delta_{v_1} = \{\delta_1, \delta_2\}$, sm_{core} is transformed into the model of v_1 shown in Fig. 1f as their application conditions are satisfied by F_{v_1}. For v_2, we apply $\Delta_{v_2} = \{\delta_3\}$. To step from v_1 to v_2, we incorporate the delta sets Δ_{v_1} and Δ_{v_2} to derive the model regression delta $\Delta_{v_1,v_2} = \{\mathsf{add}\, t_2, \mathsf{rem}\, B_1, \mathsf{rem}\, B_2, \mathsf{rem}\, B_3, \mathsf{rem}\, t_6, \mathsf{rem}\, t_7, \mathsf{rem}\, t_8, \mathsf{rem}\, t_9, \mathsf{rem}\, C_3, \mathsf{rem}\, C_2, \mathsf{rem}\, t_4, \mathsf{add}\, C_4, \mathsf{add}\, t_{10}, \mathsf{add}\, t_{11}\}$.

Next, we describe higher-order delta modeling (Lity et al., 2016a), i.e., a generic formalism to specify and reason about the evolution of delta-oriented SPLs.

Modeling variability of versions of variants: SPL evolution emerges, e.g., from changing requirements (Svahnberg and Bosch, 1999; Mens et al., 2014) and affects the set of (variable) domain artifacts and, thus, the set of variants $\mathbb{V}$. As delta modeling (Clarke et al., 2015) allows for the explicit specification of differences between variants, the concept is also applicable to define the differences between versions of variants. In prior work (Lity et al., 2016a), we defined the SPL artifact evolution by lifting the concept of delta modeling (Clarke et al., 2015) to evolve

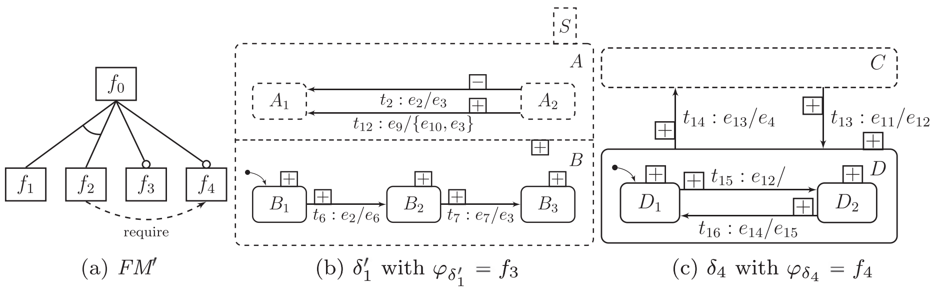


Fig. 2. (a) Evolved feature model FM'. (b) and (c) modified and added deltas.

delta models DM into their new versions DM' via higher-order deltas.

A *higher-order delta* $\delta^H = \{op^H_1, \dots, op^H_l\} \subset OP^H$ transforms an existing set of deltas Δ of a delta model DM based on the application of a set of evolution operations. An *evolution operation* $op^H \in OP^H$ specifies additions ($\mathsf{add}\,\delta$) and removals ($\mathsf{rem}\,\delta$) of deltas δ from/to a delta model. We abstract from modification operations like the exchange of the application condition of a delta and encode its modification via removal and addition. By OP^H, we refer to the set of all evolution operations defined over the universe of change operations $\mathbb{OP}$. In this paper, we further abstract from the evolution of feature models (Botterweck et al., 2010; Seidl et al., 2013; Bürdek et al., 2016; Nieke et al., 2016), but assume that for every SPL version a corresponding feature model exists building the basis for the definition of application conditions of added deltas.

The application of a higher-order delta transforms a delta model into its new version, representing an *evolution step* of the *evolution history* which we capture as *higher-order delta model*. A higher-order delta model is defined as $DM^H_{hist} = (DM^0, \Delta^H)$, where $DM^0 = (sm_{core}, \emptyset)$ represents the initial delta model to be evolved only comprising the core model, and $\Delta^H = (\delta^H_0, \delta^H_1, \dots, \delta^H_r)$ is a sequence of higher-order deltas. Similar to the delta application described above, the subsequent application of each higher-order delta $\delta^H_i \in \Delta^H$ results in the evolved version DM^i of the delta model DM^{i-1} with $i > 0$, whereas the application of the initial higher-order delta δ^H_0 completes the initial delta model $DM^0 = (sm_{core}, \Delta)$. We refer the reader to prior work (Lity et al., 2016a) for a precise description of higher-order delta application.

Example 3. The core model shown in Fig. 1b and the deltas depicted in Fig. 1c to e denote the delta model of the initial version of the sample SPL. The higher-order delta model is defined as $DM^H_{hist} = (DM^0, (\delta^H_0, \delta^H_1))$. For the initial version, we apply the higher-order delta $\delta^H_0 = \{\mathsf{add}\,\delta_1, \mathsf{add}\,\delta_2, \mathsf{add}\,\delta_3\}$, i.e., adding the three deltas, to the initial delta model $DM^0 = (sm_{core}, \emptyset)$. For its evolution to DM^1, $\delta^H_1 = \{\mathsf{rem}\,\delta_1, \mathsf{add}\,\delta'_1, \mathsf{add}\,\delta_4\}$ is applied, i.e., $DM^1 = (sm_{core}, \{\delta'_1, \delta_2, \delta_3, \delta_4\})$ holds. The modified delta δ'_1 and the new delta δ_4 are shown in Fig. 2b and c. The evolved feature model FM' is depicted in Fig. 2a, where f_4 is added as optional feature and also a requires relation between f_2 and f_4 is introduced.

3. Delta-oriented change impact analysis

To reduce the effort for testing variants and versions of variants, change impact analyses are crucial for regression testing (Yoo and Harman, 2007) guiding, e.g., retest test selection. Change impact analyses allow for reasoning about applied changes and their impact to already tested behavior of variants and also of versions of variants to be retested. One promising technique for change impact analysis in regression testing scenarios is *slicing* (Gupta et al., 1996; Agrawal et al., 1993; Bates and Horwitz, 1993; Binkley, 1998; Yoo and Harman, 2007; Tao et al., 2010), where changes to already analyzed/tested (inter)dependencies of software parts are identified.

By stepping from one variant to its subsequent one during incremental testing of a certain SPL version (cf. Section 4), we apply incremental model slicing (Lity et al., 2015) to automatically determine the changes to already tested dependencies indicating behavior to be retested. Furthermore, it is also applicable to examine changed dependencies between two consecutive versions of a variant. However, the identification of modified variants is another challenge to be explored by change impact analysis. We tackle this challenge by reasoning about the higher-order delta application to investigate changes to the variant set w.r.t. new, removed, and modified variants. We describe both strategies, i.e., the identification of change-influenced behavior as well as of versions of variants, in the next paragraphs.

Change impact analysis for variants: For the analysis of state-based models (e.g., state machines), *model slicing* (Ji et al., 2002; Kamischke et al., 2012; Androutsopoulos et al., 2013), an adaptation of program slicing (Weiser, 1981), can be applied. Based on an element, e.g., a transition, used as *slicing criterion* c, a model is reduced by abstracting from those elements/parts of the model that do not influence c. As result, we obtain a *slice* $Slice^c_{v_i}$, i.e., the reduced model of a variant v_i, preserving the execution semantics compared to the original model, but limited to the behavior influencing the given criterion c.

For the slice computation, *control and data dependencies* between model elements are determined and captured in a *model dependency graph*. The graph comprises for each model element a respective node and further directed edges connecting two nodes if a dependency between their elements exists in the model. For instance, two transitions t_j and t_k contained in parallel substate machines are synchronization dependent if t_k sends an event via an action and t_j uses it as trigger. This dependency represents that the execution of transition t_j is controlled by the execution of transition t_k. We refer the reader to the literature (Androutsopoulos et al., 2013; Kamischke et al., 2012) for a catalog of applicable dependencies.

Based on the original model, the model dependency graph, and a slicing criterion c, a slice is then computed using either *backward* or *forward slicing*. Backward slicing starts from the given model element corresponding to c and integrates those elements into the slice which potentially influence c during execution indicated by dependencies captured in the model dependency graph. In contrast, forward slicing integrates those elements into the slice which are potentially influenced by c. During the step-wise slice computation, further elements are added to a slice to ensure model well-formedness, e.g., element reachability and model connectedness. The slice computation is finished, when no further element is added to the slice, i.e., a fix-point is reached (Androutsopoulos et al., 2013). We focus on backward slicing to detect changed influences on a slicing criterion indicating retest potentials to cope with by our framework (cf. Section 4). For an overview on state-based slicing (Androutsopoulos et al., 2013) and program slicing (Xu et al., 2005), we refer the reader to the literature.

For SPLs, the application of slicing for a certain element contained in several variants every time from scratch requires much effort due to redundant computation steps. As variant-specific models share common parts, respective slices may share common parts too, which can be exploited for the slice computation. Hence, variability-aware slicing techniques (Kamischke et al., 2012; Kanning and Schulze, 2014; Lity et al., 2015) are required to reduce the slicing effort facilitating efficient SPL analysis, e.g., for reasoning about change impact.

In previous work (Lity et al., 2015), we combined model slicing (Androutsopoulos et al., 2013) and delta modeling (Clarke et al., 2015) to analyze delta-oriented SPLs. We exploit the explicit knowledge about differences between variants specified by model regression deltas to incrementally adapt the model dependency graph and already computed slices, when stepping from a variant v_{i-1} to its subsequent variant v_i to be analyzed. As result, we capture the differences between the slices of a criterion c of v_{i-1} and v_i as a *slice regression delta* $\Delta^{v_{i-1},v_i}_{Slice_c}$. Those differences represent changed dependencies w.r.t. criterion c indicating the impact of model changes applied to the original variant-specific model. We use this information in our testing framework to automatically reason about retest decisions of reusable test cases.

To adapt the model dependency graph incrementally, we reuse the change operations captured in the model regression delta. First, each element addition or removal result in a newly added node or in a node removal, respectively. Those graph changes are then used as starting point for adapting the dependencies between element nodes. For all new nodes and nodes which were connected via a dependency to a removed node, we check for new and changed element dependencies. If a dependency has to be added or removed, we also check for the respective source or target node if potentially other changes to its dependencies may arise. Otherwise, the node under consideration is skipped and the next one is examined. Therefore, we adapt only those parts of the model dependency graph which are influenced by the model changes and, thus, reduce the effort for building the dependency graph as common parts stay untouched. In addition, we exploit the changes of the model dependency graph for the incremental slice computation.

Based on the model regression delta and the dependency graph changes, we incrementally adapt existing slices of a previous variant to correspond to the current variant to be analyzed. Starting with the prior slice $Slice^c_{v_j}$ for criterion c, the captured changes are applied to adapt the slice by removing and adding respective elements from or to the slice. The adaptation is then completed by standard model slicing as new elements in the slice may introduce new dependencies for the criterion to be also added to the slice. If there is no prior slice, we apply standard model slicing to compute a new slice reusable for subsequent variants. During slice adaptation, we directly capture the differences, i.e., *slice changes*, between the prior and current slice in a *slice regression delta* $\Delta^{v_j,v_i}_{Slice_c}$. The slice changes indicate the impact of applied model changes influencing the slicing criterion referring to behavior to be retested. We refer the reader to our previous work (Lity et al., 2015) for a detailed description of incremental model slicing.

In addition to identifying slice changes between consecutive variants of an SPL version, we apply incremental model slicing (Lity et al., 2015) also for change impact analysis between versions of variants to reason about changed behavior to be retested after evolution occurs. Hence, we capture the differences between both versions of a variant also as a model regression delta, where the delta sets of the old and of the modified version are incorporated. Based on this knowledge, we apply incremental model slicing as described above to exploit the commonality between both versions and to identify new and changed behavior to guide our regression testing framework (cf. Section 4). To identify versions of variants, we automatically reason about higher-order delta application as described in the next paragraph.

Example 4. Consider the core slice for t_4 capturing all its influencing elements shown in Fig. 1b, i.e., sm_{core} as all elements potentially influence t_4 during execution. Based on Δ_{v_1} from Example 2, we obtain sm_{v_1} (cf. Fig. 1f). As the model changes have an impact on t_4, we recompute its slice depicted in Fig. 3a, where the slice changes captured in $\Delta^{v_{core},v_1}_{Slice_{t_4}}$ are denoted by $+$ for additions and $-$ for removals of elements. The variant v_1 evolves to v'_1, by applying δ^H_1 from Example 3. In Fig. 3b, the adapted slice for t_4 is shown, where compared to the previous slice version solely one new element is added, i.e., transition t_{12}.

Change impact analysis for versions of variants: To apply retest test selection after SPL evolution, change impact analysis is also required for reasoning about changes to the variant set $\mathbb{V}$, i.e., if obsolete variants are removed, existing variants are modified, or new variants are added for the new version $\mathbb{V}'$ of the SPL. Modified variants are of special interest as a modification indicates retest potentials between two versions of an already tested variant. For the identification, we exploit the differences between SPL versions captured by higher-order delta models (Lity et al., 2016a). In contrast to a naive approach, where we compare $\mathbb{V}$ with its new version $\mathbb{V}'$ in a product-by-product way to identify the performed changes, we analyze the application of higher-order deltas and derive a *variant set evolution delta* $\Delta^{evol}_{\mathbb{V}} = \{\mathsf{add}\, v_1, \ldots, \mathsf{rem}\, v_i, \ldots, \mathsf{mod}\,(v_j,\ v'_j)\}$ comprising the operations to transform $\mathbb{V}$ into $\mathbb{V}'$. Therefore, we examine the changes to variant-specific delta sets to reason about their impact as the higher-order delta application adds, modifies, or removes deltas from the delta model which is propagated to the variant-specific delta sets as well. In prior work (Lity et al., 2016a), we proposed this reasoning on a more theoretical level and abstracted from application conditions. However, to apply the change impact analysis for the evolution of real SPLs, we need to handle also application conditions and their impact on dependencies between deltas of a delta model.

In this paper, we redefine the existing reasoning about higher-order delta application by incorporating delta dependencies introduced by application conditions. We propose a delta dependency graph capturing the relations between deltas of an SPL version based on which we derive the set of variant-specific delta sets Δ_{v_i} without generating each variant-specific configuration F_{v_i}. The abstraction from configurations F_{v_i} is a benefit as we do not require all valid configurations in advance. To determine whether two deltas are somehow dependent, e.g., they are always applied together for a variant, a satisfiability check w.r.t. their application conditions is sufficient. Furthermore, as higher-order deltas describe the changes of a delta model, we pass on these changes directly to the variant-specific delta sets. Hence, the impact analysis of changes to variant-specific delta sets is independent from potential changes to feature configurations. Similar to our slicing approach (Lity et al., 2015), we exploit the changes to a delta model specified in a higher-order delta for an incremental adaptation of the delta dependency graph when the SPL evolves. As result, we obtain the changes to the graph, which in turn, are used to adapt the delta sets of the previous SPL version to correspond to its new version. We exploit the differences between the delta sets to reason about the changes from $\mathbb{V}$ to $\mathbb{V}'$ on the variant level and identify removed, added, or modified variants. In the next para-

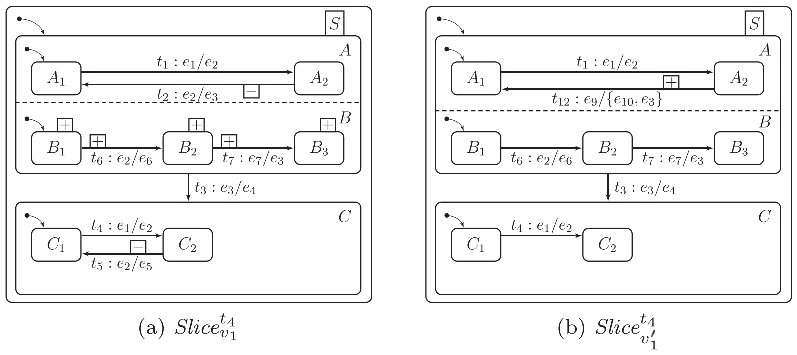


Fig. 3. Sample slices including slice regression deltas $\Delta_{Slice_{t_4}}^{v_{core},v_1}$ and $\Delta_{Slice_{t_4}}^{v_1,v_1'}$.

graphs, we define the delta dependency graph and the derivation of delta sets and afterwards describe the delta graph adaptation and change impact reasoning.

A *delta dependency graph* $DG_{DM} = (N, Dep)$ is a hypergraph containing a set of nodes N, where for each delta $\delta \in \Delta$ of a delta model DM a delta node $n_\delta \in N$ exists, and a set of dependency edges $Dep \subseteq N \times N^+$ between delta nodes denoting the dependencies between deltas $\delta \in \Delta$. To compute Dep, we introduce various *delta dependencies* classified as single- or multi-target dependencies, which we identified by investigating existing delta-oriented SPLs (Lity et al., 2016b; 2016a). A *single-target dependency* represents a one-to-one relation between two deltas captured by an edge connecting their delta nodes determined by taking their application conditions into account. We define those dependencies as follows, where we use $[\![.]\!] : F \to \mathbb{B}$ as feature selection function. A feature $f \in F$ is selected for a configuration F_v, if $[\![f]\!] = 1$ holds. By $[\![F_v]\!]$,the respective feature selection is given such that $\forall f \in F_v : [\![f]\!] = 1$ and consequently $\forall f \in (F \setminus F_v) : [\![f]\!] = 0$ holds.

Mandatory $\leftrightarrow_{man}$: Two deltas δ_i, $\delta_j \in \Delta$ are mandatory dependent denoted as $n_{\delta_i} \leftrightarrow_{man} n_{\delta_j}$, iff there is no variant such that δ_i is applied without δ_j and vice versa:

$$\neg\exists v \in \mathbb{V} : \left([\![F_v]\!] \models \neg\varphi_{\delta_i} \wedge \varphi_{\delta_j}\right) \vee \left([\![F_v]\!] \models \varphi_{\delta_i} \wedge \neg\varphi_{\delta_j}\right)$$

Exclusive $\leftrightarrow_{ex}$: Two deltas δ_i, $\delta_j \in \Delta$ are exclusive dependent represented as $n_{\delta_i} \leftrightarrow_{ex} n_{\delta_j}$, iff there is no variant such that δ_i and δ_j are both applied:

$$\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models \varphi_{\delta_i} \wedge \varphi_{\delta_j}$$

Partial Optional $\rightarrow_{pop}$: Two deltas δ_i, $\delta_j \in \Delta$ are partial optional dependent denoted as $n_{\delta_i} \rightarrow_{pop} n_{\delta_j}$, iff δ_i is independently applicable w.r.t. δ_j, but δ_j is not applicable without δ_i:

$$\exists v, v', v'' \in \mathbb{V} : \left([\![F_v]\!] \models \neg\varphi_{\delta_i} \wedge \neg\varphi_{\delta_j}\right) \wedge \left([\![F_{v'}]\!] \models \varphi_{\delta_i} \wedge \varphi_{\delta_j}\right)$$
$$\wedge\left([\![F_{v''}]\!] \models \varphi_{\delta_i} \wedge \neg\varphi_{\delta_j}\right)$$
$$\wedge\neg\exists v''' \in \mathbb{V} : \left([\![F_{v'''}]\!] \models \neg\varphi_{\delta_i} \wedge \varphi_{\delta_j}\right)$$

Complete Optional $\leftrightarrow_{cop}$: Two deltas δ_i, $\delta_j \in \Delta$ are complete optional dependent represented as $n_{\delta_i} \leftrightarrow_{cop} n_{\delta_j}$, iff both deltas are independently applicable for variants:

$$\exists v, v', v'', v''' \in \mathbb{V} : \left([\![F_v]\!] \models \neg\varphi_{\delta_i} \wedge \neg\varphi_{\delta_j}\right)$$
$$\wedge\left([\![F_{v'}]\!] \models \neg\varphi_{\delta_i} \wedge \varphi_{\delta_j}\right)$$
$$\wedge\left([\![F_{v''}]\!] \models \varphi_{\delta_i} \wedge \neg\varphi_{\delta_j}\right) \wedge \left([\![F_{v'''}]\!] \models \varphi_{\delta_i} \wedge \varphi_{\delta_j}\right)$$

In contrast, a *multi-target dependency* denotes a one-to-many relation between a source delta and some target deltas captured as edge connecting their delta nodes determined by taking the already identified single-target dependencies and in some cases also the application conditions into account. We define those dependencies as follows, where we restrict the definitions to three deltas for a better understanding, but multi-target dependencies allow for relations between more than three deltas in the same way.

Complete Alternative $\rightarrow_{calt}$: A delta $\delta_i \in \Delta$ is complete alternative dependent to δ_j, $\delta_k \in \Delta$ denoted as $n_{\delta_i} \rightarrow_{calt} \{n_{\delta_j}, n_{\delta_k}\}$, iff δ_i is solely applicable for a variant either in combination with δ_j or δ_k, but never with both deltas based on the exclusive dependency between them:

$$\left(n_{\delta_j} \leftrightarrow_{ex} n_{\delta_k}\right) \wedge \left(n_{\delta_i} \rightarrow_{pop} n_{\delta_j}\right) \wedge \left(n_{\delta_i} \rightarrow_{pop} n_{\delta_k}\right)$$
$$\wedge\, \neg\exists v \in \mathbb{V} : \left([\![F_v]\!] \models \varphi_{\delta_i} \wedge \neg\varphi_{\delta_j} \wedge \neg\varphi_{\delta_k}\right)$$

Partial Alternative $\rightarrow_{palt}$: A delta $\delta_i \in \Delta$ is partial alternative dependent to δ_j, $\delta_k \in \Delta$ represented as $n_{\delta_i} \rightarrow_{palt} \{n_{\delta_j}, n_{\delta_k}\}$, iff δ_i is solely applicable for a variant either in combination with δ_j or δ_k, but in contrast to the complete alternative dependency, where δ_j and δ_k are not applicable without δ_i, the complete optional dependent δ_k is applicable without δ_i:

$$\left(n_{\delta_j} \leftrightarrow_{ex} n_{\delta_k}\right) \wedge \left(n_{\delta_i} \rightarrow_{pop} n_{\delta_j}\right) \wedge \left(n_{\delta_i} \leftrightarrow_{cop} n_{\delta_k}\right)$$
$$\wedge\, \neg\exists v \in \mathbb{V} : \left([\![F_v]\!] \models \varphi_{\delta_i} \wedge \neg\varphi_{\delta_j} \wedge \neg\varphi_{\delta_k}\right)$$

Complete Optional Alternative $\rightarrow_{copalt}$: A delta $\delta_i \in \Delta$ is complete optional alternative dependent to δ_j, $\delta_k \in \Delta$ denoted as $n_{\delta_i} \rightarrow_{copalt} \{n_{\delta_j}, n_{\delta_k}\}$, iff δ_i is solely applicable for a variant either in combination with δ_j or δ_k, but δ_j and δ_k are also independently applicable without δ_i:

$$\left(n_{\delta_j} \leftrightarrow_{ex} n_{\delta_k}\right) \wedge \left(n_{\delta_i} \leftrightarrow_{cop} n_{\delta_j}\right) \wedge \left(n_{\delta_i} \leftrightarrow_{cop} n_{\delta_k}\right)$$
$$\wedge\, \neg\exists v \in \mathbb{V} : \left([\![F_v]\!] \models \varphi_{\delta_i} \wedge \neg\varphi_{\delta_j} \wedge \neg\varphi_{\delta_k}\right)$$

Non-Exclusive Alternative $\rightarrow_{nexalt}$: A delta $\delta_i \in \Delta$ is non-exclusive alternative dependent to δ_j, $\delta_k \in \Delta$ represented as $n_{\delta_i} \rightarrow_{nexalt} \{n_{\delta_j}, n_{\delta_k}\}$, iff δ_i is solely applicable for a variant either in combination with δ_j or δ_k, but in contrast to the complete alternative dependency, where δ_j and δ_k are not applicable together, both deltas are applicable together without δ_i:

$$\left(n_{\delta_j} \leftrightarrow_{cop} n_{\delta_k}\right) \wedge \left(n_{\delta_i} \rightarrow_{pop} n_{\delta_j}\right) \wedge \left(n_{\delta_i} \rightarrow_{pop} n_{\delta_k}\right)$$
$$\wedge\, \neg\exists v \in \mathbb{V} : \left([\![F_v]\!] \models \varphi_{\delta_i} \wedge \neg\varphi_{\delta_j} \wedge \neg\varphi_{\delta_k}\right)$$

Implication $\rightarrow_{imp}$: A delta $\delta_i \in \Delta$ is implication dependent to δ_j, $\delta_k \in \Delta$ denoted as $n_{\delta_i} \rightarrow_{imp} \{n_{\delta_j}, n_{\delta_k}\}$, iff δ_i is target of partial optional dependencies with δ_j and δ_k and both deltas are not applicable together without δ_i:

$$(n_{\delta_j} \rightarrow_{pop} n_{\delta_i}) \wedge (n_{\delta_k} \rightarrow_{pop} n_{\delta_i}) \wedge \neg\exists v \in \mathbb{V} :$$
$$([\![F_v]\!] \models \neg\varphi_{\delta_i} \wedge \varphi_{\delta_j} \wedge \varphi_{\delta_k})$$

Based on an automatically computed delta dependency graph for an SPL version, we automatically derive variant-specific delta sets by incorporating the captured delta dependencies. Therefore, we use a *variant tree*, i.e., a binary tree starting with an empty root node, where each hierarchy level represents a delta and its containment in a potential delta set. We derive a variant-specific delta set from a complete path from the root to a leaf comprising each delta either as left or right child. A delta is integrated in the variant tree, by checking for each preliminary tree path, whether the delta has to be added as left and/or right child of the last delta leaf node. Hence, we determine the potential restrictions for the delta specified by delta dependencies to predecessor delta nodes of the current tree path under consideration. A delta is added as left child, e.g., based on a mandatory dependency, denoting that the delta is contained in a delta set which is derivable from the current tree path, whereas the addition as right child represents that it is not contained in derivable variant-specific delta set, e.g., based on a exclusive or alternative dependency. If no restriction is identified, the delta is added both as left and right child to the current path, e.g., the first delta is added as left and right child as no restrictions by means of delta dependencies to already integrated deltas exist. To this end, a variant tree captures all variants of an SPL version in terms of their delta sets in a compact way.

To step to the next SPL version, a higher-order delta is applied to transform *DM* into its new version DM', where we use its captured changes to adapt the delta dependency graph incrementally in a similar way as for incremental model slicing (Lity et al., 2015). For addition/removals of deltas, we add/remove their respective nodes to/from the graph and also remove delta dependencies connected to removed nodes. Afterwards, we start from new nodes as well as from nodes influenced by removed dependencies, and re-check for changes to delta dependencies. If a changed dependency is detected, we also apply the re-check for the respective source/target node. To this end, common parts of the delta dependency graph stay untouched and we solely adapt those parts of the graph which are influenced by the higher-order delta application. We assume that existing application conditions remain constant or a delta has to be removed and added with a new application condition, e.g., due to changes in the feature model. Hence, feature model evolution (Botterweck et al., 2010; Seidl et al., 2013; Bürdek et al., 2016; Nieke et al., 2016) has no implicit impact such that the existing delta dependencies may become invalid. We record the changes to the delta dependency graph and exploit them for the incremental adaptation of the variant tree.

To reason about higher-order delta application and, hence, how the variant set $\mathbb{V}$ changes in terms of the set of delta sets Δ_{v_i}, we adapt the variant tree of the previous version and determine during adaptation whether a variant is modified, added, removed, or remains unchanged. As a first step, we deal with the removal of delta dependencies. For removed dependencies, we remove the respective restrictions of a delta node in the variant tree if one exists. In case, no restriction is left for a delta node, it is added to the variant tree and we compute its subtrees by incorporating the predecessor nodes of the current tree path under consideration. Hence, its containment in variant-specific delta sets may change, i.e., new variants are potentially derivable. Next, we remove obsolete delta nodes from the variant tree. As we already removed all restrictions in the first step which are related to those delta nodes, the subtree recomputation will result in identical subtrees for the delta node to be removed as its containment has no influence anymore to the delta set derivation. Hence, we remove a delta node from the variant tree by adding one of its identical subtrees to its parent node. We are able to directly add subtrees to the parent nodes as they are consistent to the predecessor nodes due to their computation after restrictions were removed. Third, we determine and add new node restrictions based on newly introduced delta dependencies resulting in potential removals of subtrees denoting removals of variants as their delta sets are not valid anymore. As a last step, we integrate new delta nodes as described above by checking their potential containment for each preliminary path as left and/or right child. During the adaptation, we examine the changes to the variant tree and identify whether a variant-specific delta set is added, removed, modified, or is unchanged w.r.t. its previous version.

A variant is categorized as removed if its delta set is not derivable anymore, and as unchanged if its delta set is not influenced by changes to the variant tree. In addition, a variant is modified either (1) if a previous variant is not derivable anymore, but it is extended by new/modified deltas, or (2) if its delta set contains a modified delta, but is unchanged otherwise compared to the previous version. As already mentioned, we abstract from delta modifications, but we are able to identify modified deltas such that their original versions are removed and their new versions are added during higher-order delta application. To this end, a variant is classified as new (1) if the delta (sub)set does not exists in the previous SPL version, or (2) if the previous variant is still derivable, but there is further a new delta set derivable which is an extension by new/modified deltas. We exploit this classification to reason about retest test selection after SPL evolution occurs as proposed in the next section.

Example 5. Consider the delta dependency graph $DG_{DM'}$ shown in Fig. 4a. Based on the application of δ_1^H from Ex. 3, the delta set and, thus, the dependencies between deltas change. In $DG_{DM'}$, elements without a + represent the original graph DG_{DM} for the initial SPL version captured by *DM*, whereas elements annotated with a + denote additions due to the incremental graph adaptation. In this example, no delta node or dependency is removed. The respective variant tree for $DG_{DM'}$ is depicted in Fig. 4b. Starting from the root node up to level δ_3, the four variant-specific delta sets of the initial SPL version are derivable (cf. Examples 1 and 2). Path restrictions introduced by delta dependencies are marked by a cross and the respective delta dependency responsible for the restriction. They prevent the existence of potential delta sets derivable from the path. By adding δ_4 via δ_1^H, its level is also added to the variant tree and also the modification (removal-addition encoding) of δ_1 is represented by the dashed circle. To this end, we derive six delta sets corresponding to the six feature configurations defined by FM' and identify the following variant-specific delta sets (1) $\Delta_{v_1'} = \{\delta_1', \delta_2\}$, $\Delta_{v_2'} = \{\delta_3, \delta_4\}$, and $\Delta_{v_3'} = \{\delta_1', \delta_3, \delta_4\}$ as modified, (2) $\Delta_{v_4} = \{\delta_1', \delta_2, \delta_4\}$ as well as $\Delta_{v_5} = \{\delta_4\}$ as new, and (3) the core as unchanged.

4. Product-line regression testing

To exploit the inherent reuse potential of test artifacts as well as test results between variants and versions of variants and, thus, to reduce the effort for testing SPL versions, we propose a model-based SPL regression testing framework which tackles the potential redundancy during SPL testing. Based on delta-oriented test modeling (cf. Section 2) and incremental change impact analyses (cf. Section 3), our framework allows for incremental model-based testing of one SPL version as well as subsequent SPL versions by selecting test cases for reexecution. By applying retest test selection, we validate that new behavior is implemented as specified and that already tested behavior is not influenced other than intended. In the following, we introduce the notion of model-based testing (Utting and Legeard, 2006) and afterwards describe the workflow of our regression testing framework.

Model-based testing: Model-based testing as defined by Utting and Legeard (2006) designates methods for automatic test-case generation based on test models. A *test model tm* which, in our case, is a state machine represents the behavioral specification of a *system under test*. An SUT conforms to its specification,

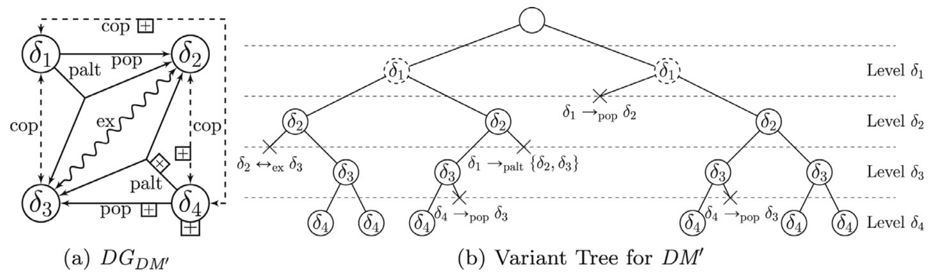

cop
palt
pop
ex
cop
cop
palt
pop
(a) $DG_{DM'}$
Level δ_1
Level δ_2
Level δ_3
Level δ_4
$\delta_1 \rightarrow_{pop} \delta_2$
$\delta_2 \leftrightarrow_{ex} \delta_3$
$\delta_1 \rightarrow_{palt} \{\delta_2, \delta_3\}$
$\delta_4 \rightarrow_{pop} \delta_3$
(b) Variant Tree for DM'

Fig. 4. (a) Delta dependency graph for SPL version DM' including changes based on the incremental dependency graph adaptation. (b) Variant tree for SPL version DM'.

if it reacts to *controllable inputs* with expected *observable outputs*. The conformance is validated, based on test cases which can be automatically generated from the test model. A *test case tc* denotes a path, i.e., an alternating sequence of states and transitions, through the test model. Based on the traversed transitions, a test case defines the sequence of controlled inputs and expected outputs for testing the corresponding SUT. For the automated generation of test cases, *coverage criteria* like *all-transitions* (Utting and Legeard, 2006) are used to control the generation process. Coverage criteria allow for the derivation of *test goals* $TG = \{tg_1, \dots, tg_k\}$, where each test goal tg_i must be covered by at least one test case. In this paper, we apply all-transitions (Utting and Legeard, 2006), i.e., every transition of a test model denotes a test goal and must be traversed by a test case. All generated test cases are collected in a *test suite* $TS = \{tc_1, \dots, tc_l\}$ and executed on the SUT to obtain test results required for our regression testing framework.

In the context of product lines, for every variant $v_i \in \mathbb{V}$ a respective test model tm_{v_i}, a set of test goals TG_{v_i}, and a test suite TS_{v_i} exist constituting the variant-specific *test artifacts* $TA_{v_i} = \{tm_{v_i}, TG_{v_i}, TS_{v_i}\}$. We use this notion to define the workflow of our SPL regression testing framework in the next paragraph.

Example 6. Consider the core state machine sm_{core} shown in Fig. 1b, now used as test model $tm_{v_{core}}$ for the core variant v_{core}. Based on all-transition coverage, we obtain the test goal set $TG_{v_{core}} = \{t_1, t_2, t_3, t_4, t_5\}$ comprising five test goals w.r.t. the contained transitions in $tm_{v_{core}}$. As we generate for each test goal a respective covering test case, the test suite is defined by $TS_{v_{core}} = \{tc_1, tc_2, tc_3, tc_4, tc_5\}$, where in this example the index of a test case correspond to the covering test goal. A sample test case $tc_5 = (S, A_1, t_1, A_2, t_2, t_3, C_1, t_4, C_2, t_5)$ is derivable, e.g., for test goal t_5, also covering the other four test goals due to their traversal via the test-case-specific path through the test model. In the end, the test model $tm_{v_{core}}$, the test goal set $TG_{v_{core}}$, and the test suite $TS_{v_{core}}$ define the test artifacts $TA_{v_{core}}$ for the core variant v_{core}.

SPL regression testing workflow: For our retest test selection technique, we combine the concepts of model-based (Utting and Legeard, 2006) and regression testing (Yoo and Harman, 2007) enabling an incremental testing workflow. To this end, SPL versions are tested subsequently by reusing test artifacts and test results of already tested variants and versions of variants. The overall workflow of our novel SPL regression testing framework is shown in Fig. 5a.

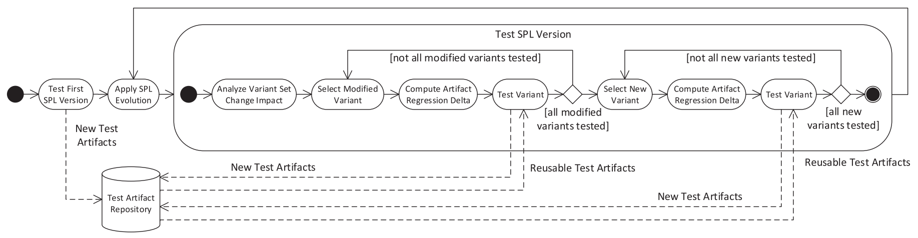
We start by testing the first, i.e., initial SPL version. For this version, no test artifacts and test results of preceding SPL versions exist to be reused. Hence, we proceed slightly different to the testing process of subsequent SPL versions to establish a test basis. The incremental sub-workflow for testing the initial version is depicted in Fig. 5b and is defined as follows.

Testing of initial SPL version. As the first step, we select the first variant $v_0 \in \mathbb{V}_0$ to be tested. Based on the application of delta modeling (Clarke et al., 2015) as test modeling formalism, the core variant is a potential option to start from. In most cases, the core denotes a minimal variant comprising the most commonality, i.e., common behavior, between all variants of the current SPL under test. Due to the commonality, the determined test artifacts have a high probability to be reused for subsequent variants under test. In contrast, the order in which variants are tested may be prescribed based on SPL prioritization techniques (Henard et al., 2014; Al-Hajjaji et al., 2016b; Lity et al., 2017) which are applied beforehand. Variants are prioritized based on their similarity or dissimilarity to each other w.r.t. their feature configurations (Henard et al., 2014; Al-Hajjaji et al., 2016b) or delta sets (Lity et al., 2017). In this paper, we abstract from the potential influence of testing orders on our framework as it is independent from a specific order and we assume a testing order to be given as input. We conduct a preliminary evaluation of the potential impact in Section 5, whereas an in-depth study of how testing orders increase the test artifact reuse and decrease the retest decisions is postponed to future work.

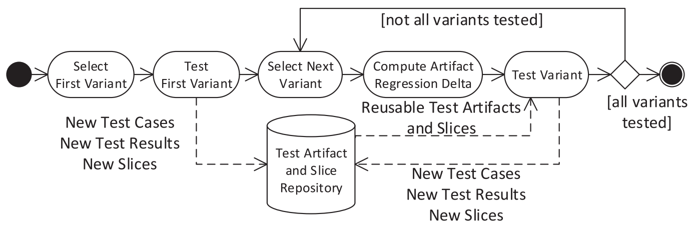
The first variant v_0 is tested by applying standard model-based testing (Utting and Legeard, 2006). Hence, we derive test goals TG_{v_0} from the test model tm_{v_0} based on all-transition coverage and generate new test cases covering those test goals to be collected in a variant-specific test suite TS_{v_0}. In addition, we apply slicing to build a basis for our incremental change impact analysis when stepping to the next variant under test. As we use all-transition coverage to derive test goals guiding the testing process, we also use test goals, i.e., transitions, as slicing criteria. To this end, we compute a slice $Slice_{v_0}^{tg_j}$ for every transition test goal $tg_j \in TG_{v_0}$ derivable from the test model tm_{v_0}. Both the test artifacts TA_{v_0} as well as all computed slices $Slice_{v_0}$ are stored in a shared *test artifact repository* after we have finished the test of variant v_0. Based on the repository, we have access to already created test artifacts and slices to be reused in upcoming testing processes of variants and versions of variants.

As the next step, we select the next variant v_{i+1} to be tested from the given testing order. To step from the previous tested variant v_i to the selected one, we need to compute the model regression delta $\Delta_{v_i,v_{i+1}}$ based on their respective delta sets Δ_{v_i} and $\Delta_{v_{i+1}}$. The regression delta is required to adapt the test artifacts for the selected variant v_{i+1} and, therefore, for testing the next variant as described in the next paragraph and shown in Fig. 5c.

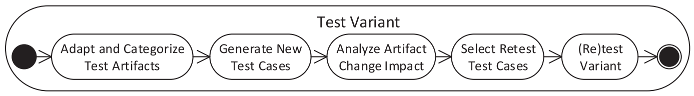
Incremental testing of a variant. Based on the test artifacts TA_{v_i} of the previous tested variant v_i restored from the test artifact repository as well as the model regression delta $\Delta_{v_i,v_{i+1}}$, we automatically adapt TA_{v_i} to get $TA_{v_{i+1}}$. First, we obtain test model $tm_{v_{i+1}}$ by applying $\Delta_{v_i,v_{i+1}}$ to tm_{v_i}. Second, we reuse the change information to derive corresponding changes to the test goal set and the test suite. For additions and removals of transitions, we add and remove the respective test goals to get $TG_{v_{i+1}}$ from TG_{v_i}. Similarly, we handle the adaptation of the test suite TS_{v_i}. Based on the path information of each test case $tc_j \in TS_{v_i}$, we examine whether it is still valid for v_{i+1}, i.e., its path is not influenced by changes and still exist in test model $tm_{v_{i+1}}$. If a test case is identified as invalid, we remove it from the test suite. In addition, we determine



(a) Incremental Version Testing Workflow



(b) Incremental SPL Testing Workflow



(c) Incremental Variant Testing Workflow

Fig. 5. Workflow of our product-line regression testing framework.

for existing test cases in the test artifact repository whether they are valid for the current variant under test. We add an existing test case to the test suite, if it is detected as valid for v_{i+1}. Hence, we obtain $TS_{v_{i+1}}$ comprising solely valid test cases from preceding variant tests.

In addition to the test suite adaptation, we categorize test cases as common in regression testing strategies (Yoo and Harman, 2007). Invalid test cases are classified as *obsolete*. Obsolete test cases are omitted for the current variant under test, but may become reusable for subsequent testing processes. All test cases captured in the adapted test suite $TS_{v_{i+1}}$ are categorized as *reusable*. From this category, we want to select a subset to be reexecuted which is also known as retest test selection problem (Yoo and Harman, 2007). In contrast, the reexecution of all reusable test cases is known as restet-all strategy (Yoo and Harman, 2007). The retest of test cases revalidate that already tested behavior is not influenced other than intended. As the last category, we have *new* test cases. We potentially require new test cases to cover new test goals in $TG_{v_{i+1}}$ as the respective elements are contained in a test model for the first time during incremental testing. Thus, the test suite $TS_{v_{i+1}} = TS^{N}_{v_{i+1}} \cup TS^{Re}_{v_{i+1}}$ combines the set of new and reusable test cases for testing variant v_{i+1}. In the end, we have a valid set of test artifacts $TA_{v_{i+1}}$ for the current variant v_{i+1} under test.

The model regression delta $\Delta_{v_i,v_{i+1}}$ capturing the differences between two consecutive tested variants further facilitate the application of change impact analysis. Each addition and removal of an element may have an impact on common behavior of both variants already tested for the previous variant v_i. We apply incremental model slicing as change impact analysis (cf. Section 3) to detect changed influences. Changed influences may have unintended side effects, e.g., based on unintended artifact interactions, indicating potentially affected behavior to be retested by selecting and, hence, retesting of reusable test cases.

For each test goal $tg_j \in TG_{v_{i+1}}$, we examine whether a prior slice $Slice^{tg_j}_{v_k}$ exist in the test artifact repository to be adapted based on previous testing processes. A test goal tg' and, hence, its respective transition which is contained in a test model for the first time does not have an existing slice in the repository. In this case, a new slice $Slice^{tg'}_{v_{i+1}}$ is computed and stored for subsequent testing steps. For the remaining test goals $tg_j \in TG_{v_{i+1}}$, we restore their last computed slice $Slice^{tg_j}_{v_k}$ from the test artifact repository and apply our incremental slicing technique to obtain the new slice $Slice^{tg_j}_{v_{i+1}}$ for the current variant v_{i+1} under test. Besides the adapted slice, we capture the differences between both slices in terms of additions and removals of slice elements as slice regression delta $\Delta^{v_k,v_{i+1}}_{Slice_{tg_j}}$. The captured differences denote changed dependencies w.r.t. the slicing criterion, i.e., test goal tg_j resulting from the applied model changes. We use this information in our regression testing framework to automatically reason about retest decisions of reusable test cases, where an empty slice regression delta indicates that the applied model changes have no impact on the slicing criterion and, thus, no retest decisions are to be made. By restoring the last computed slice, we ensure the identification of those slice differences if they exist. Nevertheless, the selection of the last computed slice depends on the given testing order and may result in redundant retest decisions for some slicing criteria in subsequent variants under test to be investigated in future work.

For reasoning about retest decisions based on the computed slice regression deltas, we define a *retest coverage criterion*. Coverage criteria like all-transition coverage (Utting and Legeard, 2006) are a meaningful scale to control distinct aspects of the testing process, e.g., test-case generation. Similar to model-based coverage criteria (Utting and Legeard, 2006), we derive retest goals $TG^{R}_{v_{i+1}} = \{rtg_1, \ldots, rtg_l\}$ by incorporating slice criteria and their slice changes captured in the respective slice regression deltas. A *retest goal* $rtg = (e_k, tg_j)$ is defined as tuple, where tg_j is the test goal used as slicing criterion for which a slice regression delta $\Delta^{v_k,v_{i+1}}_{Slice_{tg_j}}$ exists and e_k is either (1) a state/transition added to the adapted slice $Slice^{tg_j}_{v_{i+1}}$ or (2) a source/target state of a transition removed during slice adaptation. In Case (1), we validate a new dependency between element e_k and slicing criterion tg_j. In contrast, in Case (2), we validate that the removed behavior denoted by the removed transition does not remain in the implementation of the variant v_{i+1} under test.

For covering a retest goal $rtg_l \in TG^R_{v_{i+1}}$ by a reusable test case $tc \in TS^{Re}_{v_{i+1}}$, both e_k and tg_j must be traversed. A covering test case tc denotes a representative execution of variant v_{i+1} that validate that no unexpected behavior is implemented based on new or changed dependencies between both elements. To this end, each retest goal has to be covered by at least one test case to ensure retest coverage. In case, the retest goal set $TG^R_{v_{i+1}} = \emptyset$ is empty no retest decisions are to be made.

To cover all retest goals, we select reusable test cases from $TS^{Re}_{v_{i+1}}$ for a retest on the current variant v_{i+1} under test. However, some retest goals may not be covered by the current set of reusable test cases as their corresponding test model paths do not traverse both elements. In such cases, we again apply test-case generation to obtain specific *retest test cases* for covering the remaining retest goals. A retest test case explicitly denotes the representative execution of the variant under test that (re-)tests the interaction between both elements defined by a retest goal. The obtained retest test cases are generated for the current variant under test, but are potentially reusable for subsequent variants under test. In the end, the selected reusable test cases as well as the generated retest test cases build a *retest suite* $TS^R_{v_{i+1}}$ to be executed together with the new standard test cases captured in $TS^N_{v_{i+1}}$ for (re-)testing variant v_{i+1} under test. The test suites may contain several test cases that provide the same (re)test goal coverage indicating further redundancy to be reduced. Hence, there is still potential for optimization, e.g., by applying test-suite minimization (Yoo and Harman, 2007) which is out of our scope.

As the last step after test-case execution, all new and updated test artifacts $TA_{v_{i+1}}$ as well as slices $Slice_{v_{i+1}}$ are stored in the test artifact repository for subsequent testing processes. Afterwards, we select and test the next variant until no variants to be tested remain for the initial SPL version $\mathbb{V}_0$ under test.

Example 7. For testing the initial version $\mathbb{V}_0$ of our sample SPL specified by the feature model shown in Fig. 1a and the delta model depicted in Fig. 1 (cf. Examples 1 and 2), we start with the core variant $v_{core} \in \mathbb{V}_0$. The respective test artifacts $TA_{v_{core}}$ are defined in Ex. 6. For each test goal $tg_j \in TG_{v_{core}}$, we further compute a slice $Slice^{tg_j}_{v_{core}}$, e.g., the slice $Slice^{t_4}_{v_{core}}$ from Example 4 shown in Fig. 1b. Assume that we select variant $v_1 \in \mathbb{V}_0$ as next variant under test. The model regression delta Δ_{v_{core},v_1} is defined by the delta set Δ_{v_1} of v_1 from Example 2 as for the core no delta exists to be applied. Based on Δ_{v_{core},v_1}, we adapt the core test artifacts $TA_{v_{core}}$ to get TA_{v_1}. The resulting test model tm_{v_1} is shown in Fig. 1f. The test goal set TG_{v_1} is adpated by removing the test goals t_2 and t_5 as well as by adding the test goals t_6, t_7, t_8, and t_9. Due to the removal of t_2 and t_5, the only test case to be reusable for v_1 is test case tc_1 solely covering test goal t_1. The other test cases are obsolete for variant v_1 and we have to derive new test cases to ensure all-transition coverage. To identify changed influences, we apply our incremental change impact analysis. The adapted slice for test goal t_4 is shown in Fig. 3b. The slice changes captured in the slice regression delta $\Delta^{v_{core},v_1}_{Slice_{t_4}}$ are also depicted in the figure. Based on $\Delta^{v_{core},v_1}_{Slice_{t_4}}$, we derive nine retest goals, namely $rtg_1 = (A_1, t_4)$, $rtg_2 = (A_2, t_4)$, $rtg_3 = (B_1, t_4)$, $rtg_4 = (t_6, t_4)$, $rtg_5 = (B_2, t_4)$, $rtg_6 = (t_7, t_4)$, $rtg_7 = (B_3, t_4)$, $rtg_8 = (C_1, t_4)$, and $rtg_9 = (C_2, t_4)$. As we can only reuse one test case which does not cover any of those retest goals, we have to derive new retest test cases to ensure retest coverage.

Incremental testing of SPL versions. After we have finished testing the initial SPL version (cf. Fig. 5a), we apply the evolution captured by the higher-order delta δ^H_1 to step to the next SPL version $\mathbb{V}_1$ to be tested. The higher-order delta application transforms the delta model DM^0 of the previous tested SPL version to its next version DM^1 to be valid for the new SPL version under test. In this context, we apply our variant set change impact analysis (cf. Section 3) to reason about the impact of the evolution step to the set of variants $\mathbb{V}_1$ as the first step of the version testing process. Based on the analysis, we identify the removal, the addition, and the modification of variants between both versions and use this information for testing the new SPL version. We further exploit the change information to determine which variants stay unchanged during the evolution step, i.e., their delta sets are not influenced by the higher-order delta application. As we have already (re)tested those variants in the previous SPL version, unchanged variants are left out and no retest has to be applied. Hence, we solely focus on the incremental test of modified as well as new variants.

We start by testing the modified variants first. Of course, this step is skipped if no modified variants are detected by the change impact analysis. Modified variants $v^1_i \in \mathbb{V}_1$ are tested based on their previous versions $v^0_i \in \mathbb{V}_0$. Thus, a testing order is unnecessary for this testing step. Just like the testing of the first SPL version, where variants are tested incrementally, we compute the model regression delta $\Delta_{v^0_i,v^1_i}$ between both versions of the variant as a first step. Based on $\Delta_{v^0_i,v^1_i}$ capturing the differences between both versions of variant v_i, we test the modified version as described above. Therefore, we (1) adapt the test artifacts to obtain $TA_{v^1_i}$, (2) generate new test cases to ensure all-transition coverage, (3) apply incremental slicing to identify changed dependencies to be retested, and (4) select and generate test cases to ensure retest coverage. After finishing the test of a modified variant, we select the next one until all modified variants are tested.

Finally, we test all new variants of the current SPL version under test. Again, this step is skipped if no new variants are detected by the change impact analysis. In contrast to the test of modified variants, we require a testing order of the new variants to be given. Again, our approach is, in general, independent from a specific testing order. In this paper, we use the order derived from the creation of the variant tree during change impact analysis. However, similar to the testing order for the first SPL version, prioritization techniques (Henard et al., 2014; Al-Hajjaji et al., 2016b; Lity et al., 2017) are applicable. For testing the first new variant, we use the variant tree to determine its nearest partner already tested for the current SPL version under test. We determine the partner from this SPL version as new variants have more in common with similar variants from the same SPL version than with variants of previous SPL versions. In the end, we subsequently test each new variant on the basis of a previously tested new variant as described above until all new variants are tested.

After finishing the test of an SPL version, we test the next SPL version after evolution occurs. To this end, we apply the corresponding higher-order delta first and afterwards test the next version based on the test artifacts of the preceding SPL versions under test.

Example 8. Consider Example 7 again representing the example for testing the initial version of our sample SPL. For testing the next SPL version $\mathbb{V}_1$, we first apply the higher-order delta δ^H_1 from Example 3 to transform the delta model of the initial SPL version. Afterwards, we use our variant set change impact analysis to reason about the modification, addition, or removal of variants. The analysis identifies as described in Example 5 three modified variants, e.g., variant v'_1, two new variants, e.g., variant v_4, and one variant stays unchanged, i.e., the core v_{core}. No variants are removed in this evolution step. The modified variants are tested based on their previous versions. For example, the modified variant $v'_1 \in \mathbb{V}_1$ is tested based on $v_1 \in \mathbb{V}_0$. In contrast, the new variant $v_4 \in \mathbb{V}_1$ selected as first new variant under test is tested based on the nearest partner v'_1.

5. Evaluation

In this section, we present the evaluation of our regression testing framework by means of two evolving SPLs to validate its effectiveness and efficiency. We first outline shortly the prototypical implementation of our framework. Second, we describe the methodology of the evaluation and formulate research questions to be answered. Third, we introduce the evolving subject SPLs our framework is evaluated with. Finally, we present and discuss our obtained results and identify some threats to the validity of our evaluation. The experimental data as well as the prototypical implementation are available online[1].

Prototype: We realized our approach as Eclipse[2] plug-ins using the Eclipse modeling framework[3] (EMF). EMF facilitates the model-driven development of plug-ins by specifying meta-models, e.g., for delta-oriented state machines and test artifacts, from which corresponding Java source code and editors for viewing and modeling purposes are automatically generated. Based on the usage of EMF and the plug-in structure, the prototypical tool support allows for improvements and extensions in future work. Moreover, the prototype is integrated in the tool support of the research project IMoTEP[4] which focuses on efficient and effective model-based testing of evolving (dynamic) software product lines.

We further incorporated two external tools in our prototype. First, we utilized FeatureIDE (Meinicke et al., 2017) for feature modeling of the evolving subject SPLs. The version-specific feature models are required for the reasoning about the higher-order delta application (cf. Section 3). For completeness, we also derive the variant-specific feature configurations F_{v_i} for each version, but this step is not required as variants are already specified by their unique delta sets. Second, our framework requires the application of a test-case generator to automatically create test cases covering standard test goals as well as retest goals during the incremental testing of variants and versions of variants (cf. Section 4). Based on the IMoTEP project, we applied CPA/Tiger[5] (Bürdek et al., 2015) an extension of the symbolic model-checker CPA/Checker[6] for the language C. To use our test models as input for CPA/Tiger, we further implemented a transformation from our test models to C-Code by encoding the input/output event handling based on corresponding variables and their restricted access. Please note, that our prototypical implementation is independent from both tools and, therefore, allows for a flexible replacement by alternative tools with similar functionality, i.e., facilitating feature modeling and test-case generation.

Research methodology: For the documentation of the research methodology, we followed the guidelines defined by Wohlin et al. (2003) as well as Juristo and Moreno (2013). Our evaluation is defined as *controlled experiment*, where we apply our SPL regression testing framework and its prototypical tool support to two evolving subject SPLs. We use the retest-all strategy (Yoo and Harman, 2007) as *baseline* to be compared to the results obtained by our framework. To conduct the experiment, we formulate the following research questions (RQ) to be answered.

1 https://github.com/SLity/SPLregression
2 https://www.eclipse.org/
3 https://www.eclipse.org/modeling/emf/
4 http://www.dfg-spp1593.de/imotep/
5 https://www.github.com/sosy-lab/cpachecker/tree/tigerIntegration
6 https://github.com/sosy-lab/cpachecker/

RQ1 Is the reasoning about higher-order delta applications *applicable* as change impact analysis for reasoning about changes to the variant set $\mathbb{V}$ during SPL regression testing?
RQ2 Is incremental model slicing *applicable* as change impact analysis to reason about retesting of test cases in SPL regression testing?
RQ3 Do we achieve a *reduction* of test cases to be executed from our retest test selection compared to retest-all (Yoo and Harman, 2007)?
RQ4 Do we ensure *effectiveness* with our retest test selection compared to retest-all (Yoo and Harman, 2007)?
RQ5 What is the *influence* of varying testing orders on retest decisions during SPL regression testing?

The research questions **RQ1** to **RQ4** focus on the *applicability* as well as *effectiveness* of our framework, whereas **RQ5** allows for a preliminary reasoning whether there is an influence of testing orders on SPL regression testing or not. Please note that we only evaluate our retest test selection approach and the incorporated change impact analyses in our controlled experiment and abstract from the potential evaluation of the process of test modeling and test-case generation.

To answer the defined research questions, we determined both *qualitative* as well as *quantitative* data. As qualitative data, we use the delta-oriented test models of the two subject SPLs and also the other test artifacts, i.e., test goals, test cases etc., created during the execution of the experiment. By applying metrics, e.g., number of retest goals and retest test cases, to the computed test artifacts, we obtain quantitative data. We exploit the quantitative data to facilitate a *hypothesis confirmation*, where we use the defined research questions as hypothesis. To investigate whether a hypothesis can be confirmed, we apply the following data analysis.

RQ1 We assess the number of unchanged, modified, and new variants determined by the higher-order delta application reasoning when stepping to the next SPL version under test.
RQ2 We examine the number of retest goals derived by our change impact analysis (incremental model slicing). Retest goals indicate that applied model changes have an impact on already tested behavior between subsequently variants and versions of variants under test.
RQ3 We compare the number of selected and generated test cases by our framework to the number of test cases executed by retest-all (Yoo and Harman, 2007) to check for a reduction in test cases to be executed. In addition, we assess the time required for both types of change impact analyses to reason about the efficiency of our selection technique according to the cost model defined by Leung and White (Leung and White, 1991).
RQ4 We evaluate the fault detection rate of our framework compared to retest-all (Yoo and Harman, 2007). Unfortunately, the two subject SPLs do not provide a real fault history. Hence, we use simulated ones derived by taking the changes applied to a test model when stepping to a subsequent variant or version of a variant into account. For the derivation of faults, we combine the changes captured in the respective model regression deltas, i.e., added transitions as well as source and target states of removed transitions, with randomly chosen transitions from the test model. Those faults denote erroneous artifact interactions caused by changes and their impact on common behavior. We generate for each variant and version of a variant a corresponding set of those simulated faults. Depending on the size of the fault set, we create a maximum of different data sets, where each time a random selection of 10% of the faults is made to obtain random fault data sets. The set of test cases to be retested determined by our framework and the set of test

cases for retest-all (Yoo and Harman, 2007) are then executed to assess the fault detection rate.

RQ5 We examine the number of retest goals derived by our change impact analysis similar to **RQ1** as they indicate the retest potential to reason about, but we apply different testing orders to our approach. The testing orders are computed for the initial SPL versions based on existing product prioritization techniques (Al-Hajjaji et al., 2017; Lity et al., 2017; Al-Hajjaji et al., 2014; Henard et al., 2014) as well as random generation. For the random generation, we generate 100 product orders for each of the two subject SPLs. In addition, we select (1) four prioritization techniques (Al-Hajjaji et al., 2017; Al-Hajjaji et al., 2014; Henard et al., 2014) focusing on the dissimilarity between products, e.g., in terms of feature selection, to increase the coverage of elements or features, and (2) we use a testing order obtained in previous work (Lity et al., 2017), where the overall differences between subsequent variants in terms of applied change operations captured in model regression deltas are minimized. The selected techniques are applicable due to the available artifacts in this experiment required as input, i.e., delta sets and feature configurations. To this end, we apply the feature similarity Al-Hajjaji et al. (2014) and delta similarity technique (Al-Hajjaji et al., 2017) of Al-Hajjaji et al., the global maximum distance (GMD) and local maximum distance (LMD) approach from Henard et al. (2014), and from our previous work (Lity et al., 2017) the NEARIN algorithm.

Furthermore, we use the following set up for the evaluation. Although well-known in SPL testing, we do not determine samples, i.e., representative subsets, for testing the subject SPL versions. We apply our framework on the complete variant set for each SPL and its versions. For testing the initial SPL versions, we utilize the orders determined in previous work (Lity et al., 2017) based on the NEARIN algorithm which is also used to answer **RQ5**. As coverage criterion to guide standard test-case generation, we apply all-transition coverage (Utting and Legeard, 2006). Hence, we use transition test goals as slicing criteria for our incremental model slicing technique. For the slice computation, we further focus on control dependencies as proposed by Kamischke et al. (2012).

Evolving subject product lines: For the controlled experiment, we apply our framework as well as the retest-all strategy (Yoo and Harman, 2007) to two evolving SPLs (Sophia Nahrendorf and Lity, 2018), namely (1) a *Wiper SPL*, and (2) a *Vending Machine SPL*. Their original versions served already as benchmarks in the literature (Classen, 2010) and the event-based communication within those systems affect the application of our incremental change impact analysis. We examined the two systems and identified evolution scenarios in previous work (Sophia Nahrendorf and Lity, 2018) to obtain distinct versions of each subject SPL. The evolution scenarios affect our variant set change impact analysis in terms of removed, added, modified, and unchanged variants to be evaluated. The two subject SPLs and their versions are as follows:

Wiper: The original Wiper SPL (Classen, 2010) realizes the control software of a car wiper system comprising variable qualities of rain sensors and wipers. We evolved this version based on five evolution scenarios, i.e., its evolution history defines in total six versions (Sophia Nahrendorf and Lity, 2018). The scenarios include the improvement of the rain sensor and wiper, the addition of a window cleaning functionality, and a frost check. Due to the evolution, the number of variants increases from eight ($\mathbb{V}_0$) over 24 ($\mathbb{V}_3$) to 84 for the last version $\mathbb{V}_5$.

Vending machine: The original Vending Machine SPL (Classen, 2010) defines machines providing the optional selection of various beverages like coffee or tea. We evolved the original version based on seven evolution scenarios such that its evolution history defines in total eight versions (Sophia Nahrendorf and Lity, 2018). The scenarios include the introduction of variable beverage sizes, the addition of a milk counter, and the selection of different milk types for the beverages. Due to the evolution, the number of variants increases from 28 ($\mathbb{V}_0$) over 42 ($\mathbb{V}_4$) to 90 for the last version $\mathbb{V}_7$.

For the evaluation, we select solely a subset of the SPL versions from each system. Those versions denote interesting scenarios affecting our change impact analyses and, therefore, the results w.r.t. the effectiveness and efficiency of our model-based SPL regression testing framework.

Results: In Table 1, our obtained results regarding the change impact analyses as well as the retest test selection are summarized. For test goals and test cases, we provide average values w.r.t. the set of variants of the respective SPL version under test.

We answer **RQ1** by examining the categorization of the variant set change impact analysis depicted in the second column of Table 1 in the context of the applied evolution scenarios (Sophia Nahrendorf and Lity, 2018). Obviously, for both systems, all variants of the initial set of variants are classified as new. By considering the remaining versions for the Wiper SPL, we see that the reasoning about higher-order delta applications result in different categorizations. For instance, the evolution scenario for version *V*1 improves the rain sensor by incorporating a medium level for the rain detection to be handled by the wiper. This improvement influences all variants such that we detect their modifications, i.e., the original versions of the variants are not valid anymore. In contrast, for version *V*2, four variants are classified as new, whereas eight variants remain unchanged. The Wiper SPL defines the optional functionality of permanent wiping comprised in four variants in version *V*0 and *V*1. For this version (*V*2), the permanent functionality is extended by optional intensity levels. Hence, the four existing versions of the variants allowing for permanent wiping are unchanged and four new variants are introduced to realize permanent wiping based on different intensity levels. Similarly, the evolution scenarios for version *V*3 and *V*4 introduce a new optional cleaning function (*V*3) and its improvement (*V*4). As depicted in Table 1, our variant set change impact analysis provides the respective categorization with new as well as modified variants.

In version *V*0 of the Vending Machine SPL, we can either pay in euro or dollar independent from the offered beverages. When stepping to version *V*1, the evolution scenario adapts *V*0 by introducing different beverage sizes as optional selection functionality for those machines, where beverages are paid in euro. Hence, our change impact analysis identifies correctly the addition of 14 new variants. We omit the results for the remaining versions of the Vending Machine SPL as they provide no new insights for the variant set change impact analysis and also for the retest test selection compared to the discussed Wiper SPL versions.

To summarize, the results of the categorization show the applicability of our variant set change impact analysis (**RQ1**). Based on the reasoning of higher-order delta applications, we identify changes to the variant set in terms of unchanged, modified, and new versions of variants to guide the incremental testing process of our SPL regression testing framework.

We investigate the applicability of our incremental model slicing as change impact analysis (**RQ2**) by assessing the number of retest goals for both systems and their versions. As we can see, for all versions under test, we determined retest goals indicating changed influences to be retested.

For version *V*0 of the Wiper SPL, we derived a similar number of retest goals compared to the number of transition test goals. We use the number of transition test goals as approximation of

Table 1
Results of change impact analyses and retest test selection for **W**iper and **V**ending **M**achine (∅ = Average, N = New, M = Modified, U = Unchanged, T = Transition, Re = Retest, R = Reuse, O = Obsolete, S = Select).

SPL	$\Delta_{\mathbb{V}}^{evol}$	∅ Test Goals	∅ Test Cases	∅ Retest	∅ Retest	∅ Retest
	(N/M/U)	(T+Re)	(N+R+O)	(S+N)	All	Cov. (%)
W_{V0}	(8/0/0)	35.5 (17.5+18.0)	50.8 (9.8+11.5+29.5)	14.3 (7.3+7.0)	11.5	44.8
W_{V1}	(0/8/0)	104.8 (22.5+82.3)	145.9 (10.1+31.8+104.0)	18.6 (9.6+9.0)	31.8	58.8
W_{V2}	(4/0/8)	51.2 (26.3+24.8)	168.2 (3.3+39.8+125.1)	11.8 (10.0+1.8)	39.8	35.3
W_{V3}	(12/0/12)	92.8 (30.3+62.5)	249.3 (5.7+49.7+193.9)	28.1 (23.4+4.7)	49.7	65.9
W_{V4}	(0/12/12)	83.0 (32.3+50.7)	358.4 (2.5+50.1+305.8)	28.8 (27.0+1.9)	50.1	78.3
VM_{V0}	(28/0/0)	55.9 (14.7+41.2)	47.9 (2.3+7.0+38.5)	5.6 (4.1+1.5)	7.0	75.6
VM_{V1}	(14/0/28)	74.8 (17.9+56.9)	95.1 (3.4+14.5+77.2)	7.0 (4.0+3.0)	14.5	49.4

the model size. Hence, for *V*0, our change impact analysis identifies rather less changed influences during the incremental testing process of the initial variant set. One factor for the low number of retest goals may be the given testing order. Incremental model slicing is dependent on the model regression delta capturing the differences between subsequent tested variants. The size of model regression deltas between variants to be tested are in turn dependent on the order of variants given as input to our framework. We preliminary examine the impact of testing orders on the derivation of retest goals based on **RQ5**.

For version *V*1 of the Wiper SPL, we see an increased number of retest goals. The improvement of the wiper and sensor qualities by handling a medium level of rain, has a large influence on common behavior between subsequent tested versions of variants. We detect the impact of those changes represented by the high number of retest goals. The same holds for the remaining versions of the Wiper SPL. In Table 1, the average values are computed based on the complete variant set of an SPL version under test. However, when we focus on the average values for those variants which are either modified or new, we see that the changes made based on the evolution step have a large impact in terms of derivable retest goals, i.e., 74.5 for *V*2, 124.9 for *V*3, and 101.3 for *V*4, indicating influenced behavior to be retested.

For versions (*V*0) and (*V*1) of the Vending Machine SPL, the number of retest goals are higher compared to the numbers of the Wiper SPL even though the average model size is slightly smaller. In version *V*0, this is caused due to the offered beverages and their combinations representing the main behavior for the variants which is exchanged when stepping between subsequent variants under test. The exchange always has a high impact on common behavior indicated by the number of retest goals. In version *V*1, different sizes for the offered beverages are introduced. Again, we see a high impact by considering the retest goal numbers. The average value w.r.t. the complete variant set is 56.9, but for the 14 new variants, we determine 170.7 retest goals on average.

To summarize, the amount of retest goals shows that our incremental model slicing is applicable as change impact analysis technique (**RQ2**). The retest goals capture changed influences of the slicing criteria to be retested by selecting reusable test cases for reexecution. Furthermore, the testing order may affect the change impact analysis. We provide a preliminary evaluation of the impact of testing orders by assessing **RQ5**.

To answer **RQ3**, we compare the set of test cases selected and generated by our regression testing framework to the retest-all strategy (Yoo and Harman, 2007). Except for version *V*0 of the Wiper SPL, our framework determines less test cases to be reexecuted for both systems and their SPL versions under test. We mainly achieve this reduction by exploiting the results of the variant set change impact analysis, where we identify unchanged variants for which no retest has to be applied. In contrast, for the retest-all strategy, those versions of variants are also tested by reexecuting all their reusable test cases.

By comparing solely the number of test cases to be retested for modified and new versions of variants, our framework selects just as many test cases as the retest-all strategy. There are also some cases, where our framework selects slightly more. However, our retest test selection is guided based on the retest coverage criterion. We achieve complete retest goal coverage for each version of a variant and, hence, for each SPL version under test. In contrast, the retest-all strategy does not ensure the complete coverage of all retest goals as shown in the last column of Table 1. We deduce that the effort for guaranteeing retest coverage for the retest of modified and new versions of variants is acceptable, especially, as we do not retest unchanged variants.

Furthermore, when we examine the set of test cases to be retested determined by our framework, we further see that more reusable test cases are selected compared to newly generated retest test cases to ensure retest coverage. Except for version *V*0 and *V*1 of the Wiper SPL, we only require a small number of newly generated test cases. Those retest test cases represent specific execution scenarios for testing the artifact interactions defined by retest goals. The test-case generation requires further effort at this point, but test cases are only generated once and are potentially reusable and, therefore, selectable for a retest in subsequent testing processes of variants and versions of variants.

Besides the reduction of test cases to be reexecuted, the time required for change impact analyses are important to assess the efficiency of our test selection technique according to the cost model defined by Leung and White (1991). Therefore, we measured the time of the application of incremental model slicing during the incremental testing process of all variants and versions of variants, where we omit unchanged variants from the average computation as no analysis was required. On average, the slicing process took for the Wiper SPL 126ms and for the Vending Machine SPL 437ms. In addition, we measured the time of the reasoning process during higher-order delta application when stepping to next SPL version under test. The reasoning required 45*ms* for the Wiper SPL and 50ms for the Vending Machine SPL on average w.r.t. the evolution steps under consideration. We deduce that our change impact analyses are efficiently applicable as less than seconds are required for the analysis. Of course, this time will increase for larger and more complex systems as well as evolution steps. Please note that we omit the time required for the retest test generation as our approach is independent from a concrete test-case generator and, hence, the generation time will vary depending on the applied test-case generator.

In summary, the results show that our regression testing framework allows for a reduction of the test effort by reducing the number of reexecuted test cases (**RQ3**). Compared to retest-all, our framework further ensures retest goal coverage. However, the set of test cases to be retested may contain some redundancy in terms of standard as well as retest test goal coverage and can potentially further be reduced by applying test-suite minimization techniques (Yoo and Harman, 2007).

Table 2
Results of the framework effectiveness for **W**iper and **V**ending **M**achine compared to retest-all (∅ = Average).

SPL	∅ Faults	# Fault Sets	Size Fault Sets	Framework	Retest-All
				∅ Alive/∅ Dead	∅ Alive/∅ Dead
W_{V0}	37.4	22	3	0.3/2.7	1.5/1.5
W_{V1}	112.5	26	3	0.1/2.9	1.3/1.7
W_{V2}	167.5	50	6	0.5/5.5	3.3/2.7
W_{V3}	318.5	57	6	0.5/5.5	1.9/4.1
W_{V4}	366.3	100	21	3.6/17.4	7.0/14.0
VM_{V0}	41.0	30	4	0.2/3.8	1.0/3.0
VM_{V0}	170.5	85	9	0.8/8.2	4.5/4.5

Table 3
Results of the impact of product prioritization on the testing process for **W**iper and **V**ending **M**achine (∅ = Average).

Prioritization	∅ Test Goals	∅ Transition Test Goals	∅ Retest Test Goals	∅ Changes
	W_{V0}/VM_{V0}	W_{V0}/VM_{V0}	W_{V0}/VM_{V0}	W_{V0}/VM_{V0}
NEARIN Lity et al. (2017)	35.5/55.9	17.5/14.7	18.0/41.2	25.0/80.0
GMD Henard et al. (2014)	77.8/126.4	17.5/14.7	60.3/111.6	90.0/269.0
LMD Henard et al. (2014)	75.9/113.4	17.5/14.7	58.4/98.7	82.0/271.0
Feature Al-Hajjaji et al. (2014)	70.8/99.3	17.5/14.7	53.3/84.5	63.0/177.0
Delta Al-Hajjaji et al. (2017)	76.6/98.0	17.5/14.7	59.1/83.3	78.0/185.0
Random	64.4/113.4	17.5/14.7	46.9/98.7	60.8/210.1

We summarize the results of the fault detection evaluation (**RQ4**) in Table 2. The ratio of undetected (alive) and detected (dead) faults is represented by average values w.r.t. the set of tested variants of the respective SPL version under test, i.e., unchanged versions of variants are not taken into account.

We can see that our framework has, for both systems and their versions, a good fault detection rate. Furthermore, compared to retest-all (Yoo and Harman, 2007), our technique performs better. Nevertheless, we must relativize the result when considering the results for the efficiency investigation (cf. **RQ3**) also shown in Table. 1. As described above, we retest just as many test cases as retest-all to detect the faults representing random erroneous artifact interactions. However, for modified and new versions of variants, where we select less test cases than retest-all, we still have a better detection rate. Thus, our approach allows for a good detection rate with a reduced set of test cases to be retested. In addition, we assume that the small number of newly generated retest test cases to ensure retest coverage increases the fault detection rate as the set of reusable test cases selected by our framework is a subset of those test cases selected for retest-all.

To summarize, our regression testing framework provides a good fault detection capability. Compared to retest-all, the results show that our framework is more effective in finding erroneous artifact interactions. The undetected faults mainly belong to artifact interactions seeded in model parts, where changes had no impact and, hence, are not identified by our approach.

We summarize the results of the influence of product prioritization techniques on our framework (**RQ5**) in Tab. 3. We provide the average values w.r.t. the set of variants under test of the initial version of all test goals, transition test goals, retest goals, and overall changes, i.e., the sum of changes between variants captured in the model regression delta.

We can see that the varying testing orders computed by applying the different prioritization techniques (Al-Hajjaji et al., 2014; Al-Hajjaji et al., 2017; Henard et al., 2014; Lity et al., 2017) have an influence on the number of retest goals to be covered. The NEARIN algorithm (Lity et al., 2017) which was also used for the controlled experiment provides the smallest number of retest goals compared to the other techniques. This algorithm focuses on the minimization of the number of overall changes between variants to be tested. In contrast, the other applied techniques compute orders on a dissimilarity strategy to increase, e.g., the feature coverage. By taking all results into account, we derive the tendency that the smaller the number of changes, the smaller is the number of retest goals.

Summarizing, the investigation of **RQ5** represents a preliminary evaluation of the impact of testing orders on incremental SPL testing. Our framework performs better in terms of less retest goals to be covered if the changes between variants are reduced to a minimum, but this may prevent from an early fault detection as increased by following a dissimilarity strategy (Al-Hajjaji et al., 2014; Al-Hajjaji et al., 2017; Henard et al., 2014). In future experiments, we have to investigate whether the tendency we derived is confirmed and how we can optimize the potential trade-off between early fault detection and efficient regression testing.

Threats to validity: For our framework and its evaluation, the following threats to validity arise.

The test modeling step is a potential threat as due to varying interpretations of requirements and, thus, of a systems' behavior, this step may result in different test models influencing model-based testing. However, this problem applies in general for model-based testing (Utting and Legeard, 2006) and is not a specific threat for delta-oriented test modeling. To cope with this threat, we compared our re-engineered models of the original versions of the subject SPLs with the original documented models (Classen, 2010) to validate that both instances specify the same behaviors and defined the evolution scenarios based on the validated models.

For the evaluation of the effectiveness, we require faults detectable by executing test cases. We are unaware of real faults for the two subject SPLs. Hence, we created simulated faults denoting erroneous artifact interactions which is a potential threat for our evaluation. To cope with this threat and to increase the fault distribution, we select fault sets randomly to derive varying data sets for the test of a variant or version of a variant. In future work, we could improve the effectiveness evaluation by applying a model-based mutation testing framework (Aichernig et al., 2014) for the fault generation. Furthermore, the application of our framework on real model-based SPLs with an existing fault history is aspired.

The selection of the two evolving SPLs is another potential threat. The systems provide different evolution and modeling characteristics influencing the evaluation. Based on the obtained results, we assume that they are, up to a certain extent, generalizable also to other evolving model-based SPLs. However, we must substantiate this assumption by performing more experiments. Therefore, new subject SPLs should (1) be more complex and (2) provide different evolution and modeling characteristics compared to the two selected systems.

In general for model-based testing (Utting and Legeard, 2006), the applied coverage criterion is a potential threat as the choice influence several testing aspects such as the test-case generation. We applied all-transition coverage a commonly used coverage criterion (Utting and Legeard, 2006), whereas more complex criteria could be incorporated.

In this context, the choice of the test-case generator also represent a potential threat. For different generators, the obtained test cases and, hence, test suites may differ influencing the selection of test cases to be retested. Our framework is independent of a specific test-case generator such as CPA/Tiger (Bürdek et al., 2015) and, therefore, the applied generator is exchangeable by alternative tools.

As already mentioned, the given order of variants under test influences the testing of the initial SPL versions as well as the testing of new variant versions. For the initial SPL versions, we used the orders obtained in previous work on optimized testing orders (Lity et al., 2017). In contrast, for new variant versions, we use the order derived by the incremental adaptation of the variant tree. Other testing orders may influence our results as shown in the preliminary assessment of **RQ5** which could be investigated more comprehensive in future work for optimized SPL regression testing.

As last potential threat, we identified the neglection of resource factors like test-case generation and execution costs. The incorporation of resource limitations are an important aspect and could be taken into account in future experiments.

6. Related work

In this section, we discuss related work regarding (1) SPL regression testing, (2) the application of slicing for change impact analysis, and (3) SPL artifact evolution. We omit a discussion about related variability-aware slicing approaches (Kamischke et al., 2012; Kanning and Schulze, 2014; Acher et al., 2011) as well as incremental slicing techniques (Orso et al., 2001; Wehrheim, 2006; Pietsch et al., 2017) and refer to our previous work on incremental model slicing (Lity et al., 2015). For SPL testing (Oster et al., 2011; Engström, 2010; da Mota Silveira Neto et al., 2011; do Carmo Machado et al., 2012; Lee et al., 2012; Lopez-Herrejon et al., 2015) and variability modeling (Schaefer et al., 2012) in general, we also refer to the literature.

SPL regression testing: SPL regression testing is applied in the industrial context (Engström, 2010; Runeson and Engström, 2012; Runeson and Engström, 2012), for SPL integration testing (Muccini and van der Hoek, 2003; Da Mota Silveira Neto et al., 2010; Lachmann et al., 2015; 2016; 2017), for sample-based testing (Qu et al., 2007; 2008), as well as to allow for incremental testing (Uzuncaova et al., 2010; Lochau et al., 2012; 2014; Dukaczewski et al., 2013; Baller et al., 2014; Baller and Lochau, 2014; Varshosaz et al., 2015; Damiani et al., 2017). In the following, we focus on the discussion regarding the incremental testing techniques as our framework also apply the incremental strategy for efficient testing of variants and versions of variants.

One of the first incremental SPL testing strategy was proposed by Uzuncaova et al. (2010). When stepping to the next variant under test, test suites were incrementally refined. Varshosaz et al. (2015) presented a technique for test-case generation based on the application of delta modeling as test modeling formalism, where the incremental structure of delta-oriented test models is exploited to incrementally derive test cases. Baller et al. (2014) and Baller and Lochau (2014) proposed a multi-objective optimization of SPL test suites by incorporating costs and profits of test artifacts. Compared to our framework, those techniques generate and optimize SPL test suites for incremental testing of one SPL version, whereas we select test cases to be retested for incremental testing of variants and versions of variants.

Lochau et al. (2012, 2014) proposed delta-oriented component as well as integration testing by adopting the concept of delta modeling for incrementally testing an SPL. Our framework is based on their incremental workflow, especially for testing the initial SPL version, but compared to their technique, where retest decisions were made manually, we automated this process by applying change impact analysis. Dukaczewski et al. (2013) adopted the work of Lochau et al. (2012, 2014) for incremental requirements-based SPL testing. Damiani et al. (2017) also apply delta modeling as test modeling formalism for incremental model-based testing of Java programs.

In contrast to the application of regression testing for testing purposes, Heider et al. (2012) exploits regression testing to facilitate the analysis of changes to the variability model to reason about how the set of variants is affected by the changes. The change information is then propagated to the domain engineer to reduce the error potential when evolving the variability model to correspond to, e.g., changing requirements.

Slicing for change impact analysis: Slicing is mainly applied for change impact analysis in white-box regression testing (Agrawal et al., 1993; Jeffrey and Gupta, 2006; Bates and Horwitz, 1993; Gupta et al., 1996; Binkley, 1998; Tao et al., 2010; Acharya and Robinson, 2011). Agrawal et al. (1993) introduced three types of slices that comprise the executed statements for a test case. A retest of a test case is required if at least one statement in a slice is modified. Jeffrey and Gupta (2006) exploit this technique and further incorporate test-case prioritization to improve the retesting. Bates and Horwitz (1993) proposed slicing on program dependence graphs and examine slice isomorphism to reason about retest. Gupta et al. (1996) detect affected def-use pairs by applying program slicing and select test cases to retest the affected pairs. Tao et al. (2010) facilitate change impact analysis for object-oriented programs by including the logical hierarchy of object-oriented software in their slicing technique. Acharya and Robinson (2011) evaluated static program slicing in an industrial context and based on their investigation they came up with a new approach that can scale for industrial code bases. In contrast to those techniques, we apply incremental model slicing for change impact analysis integrated in model-based regression testing. We refer to Binkley (1998) for an overview of program slicing applied for change impact analysis and to Li et al. (2012) for an overview on code-based change impact analysis in general.

Slicing is also applied in the context of model-based regression testing, where Korel et al. (2002) as well as Ural and Yenigün (2013) use dependence analysis to reason about the retest of test cases. Both approaches are similar to our change impact analysis as model differences represent the starting point of the analysis. However, we compute slices for model elements and identify changes to their execution dependencies, and their dependence analysis is applied on the model path of a test case resulting in different retest decisions and, therefore, in a different selection of test cases to be reexecuted.

SPL artifact evolution: Svahnberg and Bosch (1999) already derived a general categorization of SPL evolution in 1999 by investigating two industrial case studies. They also proposed a guideline to deal with SPL evolution and its categorization. Recent work on SPL evolution is applied for different artifacts such as requirements, features, source code etc. Botterweck and Pleuss (2014), Bosch (2000), Montalvillo and Díaz (2016) or in a more global way (Borba et al., 2012; Bosch, 2002). In the following, we present and discuss related techniques for solution space evolution. As we abstract from problem space evolution mainly handled based on feature model evolution (Gamez and Fuentes, 2011; Pleuss et al., 2012; Seidl et al., 2013; Bürdek et al., 2016; Nieke et al., 2016), we refer the reader to the literature.

For solution space evolution, existing approaches adopt present variability modeling techniques such as delta modeling (Haber et al., 2012; Seidl et al., 2014; Lima et al., 2013; Dhungana et al., 2010; Lity et al., 2016a), feature-/aspect-oriented programming (Apel et al., 2005; Figueiredo et al., 2008; Alves et al., 2007), or annotative modeling (Lity et al., 2018). Seidl et al. (2014) and Haber et al. (2012) adopted delta modeling to allow for modeling the variability and evolution of SPL domain artifacts by the same means. Haber et al. (2012) support the evolution of delta-oriented architectures using specific change operations w.r.t. the delta set, whereas Seidl et al. (2014) introduce a generic formalism based on the definition of configuration and evolution deltas. Compared to higher-order delta modeling, both techniques do not facilitate change impact analysis of applied evolution steps. Lima et al. (2013) and Dhungana et al. (2010) derive deltas to identify differences between SPL versions and exploit this information for handling, i.e., detecting and resolving, conflicts introduced during evolution. Both techniques do not build on an integrated modeling formalism such as higher-order delta modeling. Apel et al. (2005) exploits the benefits of feature- as well as aspect-oriented programming by combining both approaches to handle SPL evolution and apply its formalism to FeatureC++ as an example. Alves et al. (2007) and Figueiredo et al. (2008) also use aspects for evolving the code base of an SPL. Compared to higher-order delta modeling which is artifact-independently applicable, the three techniques focus on source code and are hard to adapt for other domain artifacts.

In previous work (Lity et al., 2018), we proposed an extension for annotative modeling (Schaefer et al., 2012) called 175% modeling such that a 175% model comprises all variant-specific models of all SPL versions. Model elements are not only annotated by variability information, but also by version information to reason about for which variant or version of a variant the element is valid. 175% modeling is also a generic formalism to capture the variability and the evolution of an SPL by the same means, but compared to higher-order delta modeling does not facilitate change impact analysis to reason about changes to the variant set. However, we defined a bidirectional transformation between 175% models and higher-order delta models to exploit the benefits of both formalisms.

Neves et al. (2015) define templates for the safe evolution of SPL artifacts. Based on those templates, the application of evolution operations do not influence the behavior of existing variants other than intended. Compared to higher-order delta modeling, their technique does not provide an integrated modeling of evolution steps and also does not capture the evolution history.

SPL artifact evolution is further facilitated by applying traceability mechanisms between domain artifacts (Ajila and Kaba, 2008). In contrast to higher-order delta modeling, where evolution steps are explicitly captured, Ajila and Kaba (2008) abstract from a formalism allowing for the explicit specification of evolution steps.

7. Conclusion

In this paper, we have proposed a model-based SPL regression testing framework for incremental testing of SPL versions. Based on the extensive reuse of test artifacts as well as test results of preceding testing processes of SPL versions, we tackled the testing redundancy and, hence, reduced the overall testing effort by selecting reusable test cases for retesting variants and versions of variants. Therefore, we applied delta-oriented test modeling to capture the variability of variants and SPL versions by the same means. We exploit the explicit specification of differences between variants and SPL versions to allow for two types of automated change impact analysis. When stepping from one variant under test to a subsequent variant or its version, we use incremental model slicing to determine the influences of applied model changes. The identified changed influences captured in slice regression deltas facilitate the definition of a retest coverage criterion to be ensured by selecting reusable test cases for reexecution. In addition, we reason about the higher-order delta application when stepping from one SPL version under test to the next one. The reasoning allows for the identification of changes to the variant set in terms of modified, removed, added, and unchanged variants. We exploit this information to focus on the retesting of modified and new variant versions. In the end, we have evaluated our framework and its prototypical tool support in a controlled experiment regarding its efficiency and effectiveness by means of two evolving SPLs.

Our evaluation shows positive results. Both types of change impact analysis are efficiently applicable to determine retest potentials during incremental testing of variants and versions of variants. Based on the analysis results, we achieve a reduction of test cases to be executed compared to retest-all (Yoo and Harman, 2007). Furthermore, our selection provides a good fault detection rate. To this end, our framework facilitates the reduction of the increasing test effort of evolving SPLs enabling a more targeted use of limited test resources.

To generalize our obtained results, we want to apply our regression testing framework to other model-based SPLs from academia and industry. Especially, the application to real model-based SPLs is aspired to investigate its practicability. Furthermore, we want to examine the influence of testing orders on the framework by identifying characteristics of those orders improving the efficiency in terms of decreasing retest decisions to be made. In this context, the incorporation of test-suite minimization techniques (Yoo and Harman, 2007) is also imaginable to reduce the overall testing effort by reducing the number of executed test cases. To increase the applicability of our approach in practice, we could incorporate other test-model types, e.g., software architectures, to facilitate its application also in other testing phases such as integration testing. For this extension, our selection technique as well as the change impact analysis between variants and versions of variants has to be adapted to handle the new test-model type and corresponding test artifacts. In this context, the incorporation of other variability implementation techniques (Schaefer et al., 2012) for test modeling is also possible, where a transformation between, e.g., annotative and delta modeling, has to be defined (Fenske et al., 2014; Lity et al., 2018). To this end, our framework allows for a more general application in SPL regression testing, but a corresponding engineering and implementation effort is required for the adaptation.

Acknowledgments

This work was partially supported by the German Research Foundation under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

Acharya, M., Robinson, B., 2011. practical change impact analysis based on static program slicing for industrial software systems. In: Proceedings of the ICSE'11. ACM, pp. 746–755.
Acher, M., Collet, P., Lahire, P., France, R.B., 2011. Slicing feature models. In: Proceedings of the ASE'11, pp. 424–427.
Agrawal, H., Horgan, J.R., Krauser, E.W., London, S., 1993. Incremental Regression Testing. In: Proceedings of the ICSM '93. IEEE Computer Society, pp. 348–357.
Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W., Schlick, R., Tiran, S., 2014. Killing strategies for model-Based mutation testing. Softw. Test. Verif. Reliab. 25 (8), 716–748.

Ajila, S.A., Kaba, A.B., 2008. Evolution support mechanisms for software product line process. J. Sys. Soft. 81 (10), 1784–1801.

Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., Saake, G., 2016. IncLing: efficient product-line testing using incremental pairwise sampling. In: Proceedings of the GPCE'16. ACM, pp. 144–155.

Al-Hajjaji, M., Lity, S., Lachmann, R., Thüm, T., Schaefer, I., Saake, G., 2017. Delta-oriented product prioritization for similarity-based product-line testing. In: Proceedings of the VACE'17, pp. 34–40.

Al-Hajjaji, M., Thüm, T., Lochau, M., Meinicke, J., Saake, G., 2016. Effective product-Line testing using similarity-Based product prioritization. Software & Systems Modeling 1–23.

Al-Hajjaji, M., Thüm, T., Meinicke, J., Lochau, M., Saake, G., 2014. Similarity-based prioritization in software product-line testing. In: Proceedings of the SPLC'14, pp. 197–206.

Alves, V., Matos Jr., P., Cole, L., Vasconcelos, A., Borba, P., Ramalho, G., 2007. Extracting and evolving code in product lines with aspect-oriented programming. In: Trans. Aspect-Oriented Soft. Dev., pp. 117–142.

Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., Tratt, L., 2013. State-based model slicing: A Survey. ACM Comput. Surv. 45 (4), 53:1–53:36.

Apel, S., Leich, T., Rosenmüller, M., Saake, G., 2005. Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution. In: Proceedings of the RAM-SE'05-ECOOP'05.

Baller, H., Lity, S., Lochau, M., Schaefer, I., 2014. Multi-objective test suite optimization for incremental product family testing. In: Proceedings of the ICST'14, pp. 303–312.

Baller, H., Lochau, M., 2014. Towards incremental test suite optimization for software product lines. In: Proceedings of the FOSD'14. ACM, pp. 30–36.

Bates, S., Horwitz, S., 1993. Incremental program testing using program dependence graphs. In: Proceedings of the POPL'93. ACM, pp. 384–396.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: a literature review. Inf. Syst. 35 (6), 615–636.

Binkley, D., 1998. The application of program slicing to regression testing. Inf. Softw. Technol. 40 (1112), 583–594.

Borba, P., Teixeira, L., Gheyi, R., 2012. A theory of software product line refinement. Electronic Notes in Theoretical Computer Science 455, 2–30.

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Pearson.

Bosch, J., 2002. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In: Proceedings of the SPLC'02. Springer, pp. 257–271.

Botterweck, G., Pleuss, A., 2014. Evolution of Software Product Lines. Springer, pp. 265–295.

Botterweck, G., Pleuss, A., Dhungana, D., Polzer, A., Kowalewski, S., 2010. EvoFM: feature-driven planning of product-line evolution. In: Proceedings of the PLEASE'10. ACM, pp. 24–31.

Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A., 2016. Reasoning about product-Line evolution using complex feature model differences 23 (4), 687–733.

Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D., 2015. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In: Proceedings of the FASE'15. In: LNCS, 9033. Springer, pp. 84–99.

do Carmo Machado, I., McGregor, J.D., Santana de Almeida, E., 2012. Strategies for testing products in software product lines. SIGSOFT Softw. Eng. Notes 37 (6), 1–8.

Clarke, D., Helvensteijn, M., Schaefer, I., 2015. Abstract delta modelling. Math. Struct. Comp. Sci. 25 (3), 482–527.

Classen, A., 2010. Modelling with FTS: A Collection of Illustrative Examples. Technical Report. PReCISE Research Center, Univ. of Namur.

Czarnecki, K., Eisenecker, U., 2000. Generative Programming: Methods, Tools, and Applications. Addison Wesley.

Da Mota Silveira Neto, P., do Carmo Machado, I., Cavalcanti, Y., de Almeida, E., Garcia, V., de Lemos Meira, S., 2010. A Regression Testing Approach for Software Product Lines Architectures. In: Proceedings of the SBCARS'10, pp. 41–50.

Damiani, F., Faitelson, D., Gladisch, C., Tyszberowicz, S., 2017. A novel model-Based testing approach for software product lines. Softw. Syst. Model. 16 (4), 1223–1251.

Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T., 2010. Structuring the modeling space and supporting evolution in software product line engineering. J. Sys. and Soft. 83 (7), 1108–1122.

Dukaczewski, M., Schaefer, I., Lachmann, R., Lochau, M., 2013. Requirements-based Delta-oriented SPL Testing. In: Proceedings of the PLEASE'13, pp. 49–52.

Engström, E., 2010. Exploring Regression Testing and Software Product Line Testing - Research and State of Practice. Lund University Lic dissertation.

Fenske, W., Thüm, T., Saake, G., 2014. A Taxonomy of Software Product Line Reengineering. In: Proceedings of the VaMoS'14. ACM, New York, NY, USA, pp. 4:1–4:8.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., et al., 2008. Evolving software product lines with aspects. In: Proceedings of the ICSE'08, pp. 261–270.

Gamez, N., Fuentes, L., 2011. Software product line evolution with cardinality-based feature models. In: Proceedings of the ICSR'11, pp. 102–118.

Gupta, R., Harrold, M.J., Soffa, L., 1996. Program slicing-Based regression testing techniques. Softw. Test., Verif. Reliab. 6, 83–112.

Haber, A., Rendel, H., Rumpe, B., Schaefer, I., 2012. Evolving Delta-Oriented Software Product Line Architectures. In: Proceedings of the Monterey Workshop. Springer, pp. 183–208.

Harrold, M.J., 2000. testing: a roadmap. In: Proceedings of the ICSE'00. ACM, pp. 61–72.

Heider, W., Rabiser, R., Grünbacher, P., Lettner, D., 2012. Using regression testing to analyze the impact of changes to variability models on products. In: Proceedings of the SPLC'12. ACM, pp. 196–205.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Traon, Y.L., 2014. Bypassing the combinatorial explosion: using similarity to generate and prioritize T-Wise test configurations for software product lines. IEEE Trans. Software Eng. 40 (7), 650–670.

Jeffrey, D., Gupta, R., 2006. Test case prioritization using relevant slices. In: Proceedings of the COMPSAC'06, 1, pp. 411–420.

Ji, W., Wei, D., Zhi-Chang, Q., 2002. Slicing hierarchical automata for model checking UML statecharts. In: Proceedings of the ICFEM'02, pp. 435–446.

Johansen, M.F., Haugen, O., Fleurey, F., 2012. An algorithm for generating t-wise covering arrays from large feature models. In: Proceedings of the SPLC'12. ACM, pp. 46–55.

Juristo, N., Moreno, A.M., 2013. Basics of Software Engineering Experimentation. Springer Science & Business Media.

Kamischke, J., Lochau, M., Baller, H., 2012. Conditioned model slicing of feature-annotated state machines. In: Proceedings of the FOSD'12, pp. 9–16.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis Feasibility Study. Technical Report. CMU SEI.

Kanning, F., Schulze, S., 2014. Program slicing in the presence of preprocessor variability. In: Proceedings of the ICSME'14, pp. 501–505.

Korel, B., Tahat, L., Vaysburg, B., 2002. Model-based regression test reduction using dependence analysis. In: Proceedings of the ICSM'02, pp. 214–223.

Kowal, M., Tschaikowski, M., Tribastone, M., Schaefer, I., 2015. Scaling size and parameter spaces in variability-aware software performance models. In: Proceedings of the ASE'15, pp. 407–417.

Lachmann, R., Beddig, S., Lity, S., Schulze, S., Schaefer, I., 2017. Risk-based integration testing of software product lines. In: Proceedings of the VaMoS'17. ACM, pp. 52–59.

Lachmann, R., Lity, S., Al-Hajjaji, M., Fürchtegott, F., Schaefer, I., 2016. Fine-grained test case prioritization for integration testing of delta-oriented software product lines. In: Proceedings of the FOSD'16. ACM, pp. 1–10.

Lachmann, R., Lity, S., Lischke, S., Beddig, S., Schulze, S., Schaefer, I., 2015. Delta-oriented test case prioritization for integration testing of software product lines. In: Proceedings of the SPLC'15, pp. 81–90.

Lee, J., Kang, S., Lee, D., 2012. A survey on software product line testing. In: Proceedings of the SPLC'12. ACM, pp. 31–40.

Leung, H.K.N., White, L., 1991. A cost model to compare regression test strategies. In: Proceedings of the Conference on Software Maintenance 1991, pp. 201–208.

Li, B., Sun, X., Leung, H., Zhang, S., 2012. A survey of codebased change impact analysis techniques. Softw. Test. Verif. Reliab. 23 (8), 613–646.

Lima, G., Santos, J., Kulesza, U., Alencar, D., Fialho, S.V., 2013. A delta oriented approach to the evolution and reconciliation of enterprise software products lines. In: Proceedings of the ICEIS'13, pp. 255–263.

Linden, F.J.v.d., Schmid, K., Rommes, E., 2007. Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer.

Lity, S., Al-Hajjaji, M., Thüm, T., Schaefer, I., 2017. Optimizing product orders using graph algorithms for improving incremental product-line analysis. In: Proceedings of the VaMoS'17. ACM, pp. 60–67.

Lity, S., Baller, H., Schaefer, I., 2015. Towards incremental model slicing for delta-oriented software product lines. In: Proceedings of the SANER'15, pp. 530–534.

Lity, S., Kowal, M., Schaefer, I., 2016. Higher-order delta modeling for software product line evolution. In: Proceedings of the FOSD'16. ACM, pp. 39–48.

Lity, S., Morbach, T., Thüm, T., Schaefer, I., 2016. Applying incremental model slicing to product-line regression testing. In: Proceedings of the ICSR'16. Springer.

Lity, S., Nahrendorf, S., Thüm, T., Seidl, C., Schaefer, I., 2018. 175 Artifacts. In: Proceedings of the VaMoS'18. ACM, pp. 27–34.

Lochau, M., Lity, S., Lachmann, R., Schaefer, I., Goltz, U., 2014. Delta-oriented model-based integration testing of large-scale systems. J. Syst. Softw. 91, 63–84.

Lochau, M., Schaefer, I., Kamischke, J., Lity, S., 2012. Incremental model-based testing of delta-oriented software product lines. In: Proceedings of the TAP'12. Springer, pp. 67–82.

Lopez-Herrejon, R.E., Fischer, S., Ramler, R., Egyed, A., 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In: Proceedings of the ICSTW'15, pp. 1–10.

McGregor, J.D., 2001. Testing a Software Product Line. Technical Report. Carnegie Mellon University.

Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G., 2017. Mastering Software Variability with FeatureIDE. Springer.

Mens, T., Serebrenik, A., Cleve, A. (Eds.), 2014. Evolving Software Systems. Springer.

Montalvillo, L., Díaz, O., 2016. Requirement-driven evolution in software product lines: a systematic mapping study. J. Sys. Soft. 122, 110–143.

da Mota Silveira Neto, P.A., Carmo Machado, I.d., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R., 2011. A systematic mapping study of software product lines testing. Inf. Softw. Technol. 53, 407–423.

Muccini, H., van der Hoek, A., 2003. Towards testing product line architectures. In: Electronic Notes Theory Computing Science, 82, pp. 99–109. TACoS'03

Neves, L., Borba, P., Alves, V., Turnes, L., Teixeira, L., Sena, D., Kulesza, U., 2015. Safe evolution templates for software product lines. J. Syst. Soft. 106, 42–58.

Nieke, M., Seidl, C., Schuster, S., 2016. Guaranteeing configuration validity in evolving software product lines. In: Proceedings of the VaMoS'16. ACM, pp. 73–80.

Orso, A., Sinha, S., Harrold, M.J., 2001. Incremental slicing based on data-dependences types. In: Proceedings of the ICSM'01, p. 158.

Oster, S., Wübbeke, A., Engels, G., Schürr, A., 2011. A survey of model-based software product lines testing. In: Model-based Testing for Embedded Systems. CRC Press, pp. 338–381.
Pietsch, C., Kehrer, T., Kelter, U., Reuling, D., Ohrndorf, M., 2015. SiPL – a delta-based modeling framework for software product line engineering. In: ASE'15, pp. 852–857.
Pietsch, C., Ohrndorf, M., Kelter, U., Kehrer, T., 2017. Incrementally Slicing Editable Submodels. In: Proceedings of the ASE'17. IEEE Press, pp. 913–918.
Pleuss, A., Botterweck, G., Dhungana, D., Polzer, A., Kowalewski, S., 2012. Model–driven support for product line evolution on feature level. J. Syst. Soft. 85 (10), 2261–2274.
Pohl, K., Böckle, G., Linden, F.J.v.d., 2005. Software Product Line Engineering: Foundations, Principles and techniques. Springer.
Qu, X., Cohen, M., Woolf, K., 2007. Combinatorial interaction regression testing: a study of test case generation and prioritization. In: Proceedings of the ICSM'07, pp. 255–264.
Qu, X., Cohen, M.B., Rothermel, G., 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. In: Proceedings of the ISSTA'08, pp. 75–86.
Rothberg, V., Dietrich, C., Ziegler, A., Lohmann, D., 2016. Towards Scalable Configuration Testing in Variable Software. In: Proceedings of the GPCE'16. ACM, pp. 156–167.
Runeson, P., Engström, E., 2012. Regression testing in software product line engineering. Adv. Comput. 86, 223–263.
Runeson, P., Engström, E., 2012. Software Product Line Testing – A 3D Regression Testing Problem. In: Proceedings of the ICST'12. IEEE, pp. 742–746.
Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K., 2012. Software diversity: state of the art and perspectives. STTT 14 (5), 477–495.
Seidl, C., Schaefer, I., Aßmann, U., 2013. Capturing variability in space and time with hyper feature models. In: Proceedings of the VaMoS'13. ACM, pp. 6:1–6:8.
Seidl, C., Schaefer, I., Aßmann, U., 2014. Integrated management of variability in space and time in software families. In: Proceedings of the SPLC'14. ACM, pp. 22–31.
Sophia Nahrendorf, I.S., Lity, S., 2018. Applying Higher-Order Delta Modeling for the Evolution of Delta-Oriented Software Product Lines. Technical Report. TU Braunschweig - Institute of Software Engineering and Automotive Informatics.
Svahnberg, M., Bosch, J., 1999. Evolution in software product lines: two cases. J. Softw. Maint. 11 (6), 391–422.
Tao, C., Li, B., Sun, X., Zhang, C., 2010. An Approach to Regression Test Selection Based on Hierarchical Slicing Technique. In: Proceedings of the COMPSACW'10, pp. 347–352.
Tevanlinna, A., Taina, J., Kauppinen, R., 2004. Product family testing: A Survey. ACM SIGSOFT Softw. Eng. Notes 29.
Ural, H., Yenigün, H., 2013. Regression test suite selection using dependence analysis. J. Softw. Evolut. Process 25 (7), 681–709.
Utting, M., Legeard, B., 2006. Practical Model-Based testing: A Tools approach. Morgan Kaufmann Publishers Inc.
Uzuncaova, E., Khurshid, S., Batory, D., 2010. Incremental test generation for software product lines. IEEE Trans. Softw. Eng. 36 (3), 309–322.
Varshosaz, M., Beohar, H., Mousavi, M.R., 2015. Delta-Oriented FSM-Based Testing. In: Proceedings of the ICFEM'15. Springer, pp. 366–381.
Wehrheim, H., 2006. Incremental Slicing. In: Proceedings of the ICFEM'06, pp. 514–528.
Weiser, M., 1981. Program slicing. In: Proceedings of the ICSE'81, pp. 439–449.
Weiss, D.M., 2008. The product line hall of fame. In: Proceedings of the SPLC'08 395.
Wille, D., Runge, T., Seidl, C., Schulze, S., 2017. Extractive software product line engineering using model-based delta module generation. In: Proceedings of the VaMoS'17. ACM, New York, NY, USA, pp. 36–43.
Wohlin, C., Höst, M., Henningsson, K., 2003. Empirical Research Methods in Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 7–23.
Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L., 2005. A brief survey of program slicing. SIGSOFT Softw. Eng. Notes 30 (2), 1–36.
Yoo, S., Harman, M., 2007. Regression testing minimization, selection and prioritization: a survey. Softw. Test., Verif. Reliab. 22 (2), 67–120.

Sascha Lity is a Ph.D. Student at the Technische Universität Braunschweig. He finished his Master of Science in 2011 and works as research assistant at the Institute of Software Engineering and Automotive Informatics. Main interests of his research are evolving software product lines and model-based testing of variant-rich systems. He is a member of the DFG project IMoTEP.

Manuel Nieke is a Ph.D. Student at the Technische Universität Braunschweig. He was a student assistant at the Institute of Software Engineering and Automotive Informatics and finished his Master of Science in 2018. He works as research assistant at the Institute for Operating Systems and Computer Networks. Main interests of his research are distributed systems and trusted remote execution.

Thomas Thüm has a postdoc position at the Institute of Software Engineering and Automotive Informatics at the Technische Universität Braunschweig. He received his Ph.D. degree from the Otto-von-Guericke-Universität Magdeburg. His research interests are feature-oriented software development, product-line analysis and variability modeling. He is one of the main developers of FeatureIDE.

Ina Schaefer is chair of the Institute of Software Engineering and Automotive Informatics at the Technische Universität Braunschweig. She received her Ph.D. degree from the TU Kaiserslautern and worked as a postdoc at the Chalmers University of Technology in Gothenburg, Sweden. Her research interests are verification and testing methods for variant-rich and evolving software systems.