Automatic and Incremental Product Optimization for Software Product Lines

Andreas Demuth
Johannes Kepler University
Linz, Austria
Email: andreas.demuth@jku.at

Roberto E. Lopez-Herrejon
Johannes Kepler University
Linz, Austria
Email: roberto.lopez@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
Email: alexander.egyed@jku.at

***Abstract*—Software Product Lines (SPLs) have gained popularity in industry as they foster the reuse of artifacts, such as code, and reduce product development effort. Although some SPLs ensure that only valid products are configurable, those products are not necessarily optimal. For instance, they may include code that is not necessary for providing the desired functionality – often because of erroneous traceability between features and code. Such unnecessary code may be disallowed in safety critical domains, it may lead to losses in runtime performance, or it may lead to errors during later SPL evolution. In this paper, we present an approach for automatic and incremental product optimization. Our approach leverages product functionality tests to ensure that configured products do not include unnecessary artifacts – an automatic re-optimization of products after SPL evolution is performed incrementally. The evaluation results show that such a re-optimization takes only milliseconds.**

***Keywords*—Software product lines, optimization, evolution**

I. Introduction

Software Product Lines (SPLs) [1] are commonly used in industry as they foster and support software reuse, enabling quicker development of products. Individual units of functionality are typically bundled in *features* [2]. The functionality shared between all products as well as the possible points of variation (i.e., product variability) are defined in *feature models* [3] that describe feature relations and thus define the set of configurable products. Based on feature models, products are configured by selecting a subset of the available features. A selection is valid if it conforms to consistency rules imposed by the feature model and the relations between features. Based on a selection, artifacts (or *assets* [4]) associated with the selected features are combined to form products that can be delivered to customers. For SPLs in particular, those assets typically consist of source code, test scripts, or software models that are combined – transformed to code in the case of models – and compiled.

While it seems that those basic SPL concepts are easy to implement, literature has identified several serious challenges [5]. One of the biggest threats are errors in a feature model or the traceability between features and the assets that implement them. Such errors may lead to products that are correct with respect to the feature model but that cannot be compiled because of – just to name one example – missing files. Hence, checking the correctness of products is essential for successfully using SPLs [6]. *Safe composition* [2] is an important approach that addresses this common and complex issue by ensuring that all valid feature combinations lead to type-safe programs. Note that these programs may not be working as expected in practice (e.g., the desired functionality is not provided).

However, the correctness ensured by approaches such as safe composition does not mean configured products are optimal. For example, configured products may contain not only those assets (e.g., source code files) that are required for providing the desired functionality, but they may also contain additional assets. This phenomenon occurs when the association between assets and features is done incorrectly during variability modeling of the product line. In practice, product lines are often introduced when the need for customization of an existing product grows. Variability modeling is then often done by breaking down a reference system into individual features [7]. Since it is often not possible do easily determine which feature is implemented by which asset [8] – as the reference system was not developed with reuse of individual components in product lines in mind – product line designers may associate assets with common features (e.g., the root feature). In doing so, they ensure that the assets are available in a large number of products and that product configuration is safe (according to approaches such as safe composition). However, additional (i.e., unnecessary) assets may be prohibited in safety-critical domains [9] and may also affect a product in various ways. For instance, the system requirements for installing and running the delivered product may be increased and runtime performance may suffer significantly.

Moreover, as domains and market demands evolve over time, so do SPLs [5], [10]. During evolution, non-optimal products imply a higher risk of emerging errors due to potential dependencies between assets. Of course, evolving an SPL may also cause optimal products to become non-optimal. Therefore, SPL evolution may also require a re-optimization of products.

In this paper we present an automatic approach for optimizing products and removing unnecessary assets while preserving functionality. We define general conditions that have to be established for optimal products and discuss how those conditions can be established. In particular, we discuss how the approach can be applied incrementally for evolving product lines with a limited set of existing products.

II. Software Product Lines in Practice

Before we discuss common SPL issues and product optimization in detail, let us first describe basic SPL modeling

concepts, discuss how those concepts are adopted by companies in practice, and illustrate the role of product testing in SPLs.

A. Basic Concepts

Formally, a product line (PL) consists of the following parts, as shown in (1): a) feature model (FM), b) asset model (AM), c) feature-to-asset traces (FAT), and products (PR).

$$\text{PL} := \langle \text{FM}, \text{AM}, \text{FAT}, \text{PR} \rangle \tag{1}$$

1) Feature Model: The feature model, as shown in (2), contains a set of features (F) and is used for describing the different functionality a product can provide at a high level of abstraction [1]. Moreover, there are associations (FA) and additional cross-tree constraints (FC) that both express how features are related and how they should be selected during configuration. Finally, one of the existing features is selected as the models root feature.

$$\text{FM} := \langle \text{F}, \text{FA}, \text{FC}, \text{root} \in \text{F} \rangle \tag{2}$$

2) Asset Model: An asset model, as shown in (3), includes all the assets (A) which may be used to compose the actual product to be delivered. Although these assets are typically source code files, configuration information or data schemas, image files used by user interfaces, etc., abstractions of code may also be used. For example, UML models that are transformed to source code. Note that assets are generic and can represent arbitrary artifacts at any level of granularity. For instance, an asset may be defined for an entire file or also for a specific method. This view of assets stems from common product line tools and techniques that allow for the composition of individual methods or even variables in source files or design models [11], [12].

In contrast to features, assets are not organized hierarchically (i.e., there is no root asset). However, as there may be dependencies between assets (e.g., method `a` calls method `b`) or also conflicts if certain assets are present in a system at the same time (e.g., alternative implementation of a method), thus there may be asset constraints (AC) that describe such relations.

$$\text{AM} := \langle \text{A}, \text{AC} \rangle \tag{3}$$

3) Feature-to-Asset Traces: When the functionality of possible products and the assets from which those products can be built are defined, it still has to be defined which assets are used for implementing which feature. This is done via feature-to-asset traces (FAT), shown in (4). Each trace is a pair that consists of a feature and a set of assets.

$$\text{FAT} := \{\langle x \in \text{F}, y \subseteq \text{A} \rangle\} \tag{4}$$

4) Products: Finally, we define products. The configured products (PR) simply consists of individual configurations, as shown in (5). Each configuration is identified by two sets, one holding the selected features and one holding the assets from which the configured product is composed.

$$\text{PR} := \{\langle f \subseteq \text{F}, a \subseteq \text{A} \rangle | \forall x \in f : \exists \langle x, y \rangle \in \text{FAT} \Rightarrow y \subseteq a\} \tag{5}$$

B. Adoption Process

Let us now briefly discuss the typical process of SPL concept adoption. The ideal, proactive, process starts with defining common and variable functionality of desired products (i.e., the feature model) [7]. It is followed by a structured development of assets with the defined variability and systematic reuse in mind [13]. Traces between features and assets are therefore easy to identify.

However, this ideal process cannot always be applied in practice as it requires planning desired products before their development. Typically, the decision for establishing a well defined product line is made after an initial (reference) product was developed and partly adapted several times afterwards to tailor it to different customer needs [14]. While breaking down the functionality of desired products to features is not affected by the existing reference product, the already existing assets that implement those features have not been developed with product variability and systematic reuse in mind – quite contrary to the ideal SPL process discussed above. This means that defining the required feature-to-asset traceability is far from trivial and requires detailed knowledge about the reference product. Existing approaches that generate such traces from products (e.g., [15]) typically rely on the actual – and thus potentially non-optimal – asset combinations. Thus, obtaining correct traceability for complex systems that leads to optimal products remains a challenge [8].

C. Product Testing

Let us now describe how product testing is used to validate that individual products provide the expected functionality and meet customer expectations. For every feature, and also feature combinations, a specific set of test cases has to be executed successfully to ensure correct behavior. We define the set of all available test cases as TC. Inspired by [6], the association between features and tests is called feature-to-test traceability (FTT), as shown in (6).

$$\text{FTT} := \{\langle x \subseteq \text{F}, y \subseteq \text{TC} \rangle\} \tag{6}$$

The complete test suite for a product is the sum of the test cases associated with the individual features (or combinations thereof) selected in a product, as shown in (7).

$$\begin{aligned} &testSuite : \mathcal{P}(\text{F}) \rightarrow \mathcal{P}(\text{TC}), \\ &x \mapsto \{t \in \text{TC} | \exists z \subseteq x, \langle u, v \rangle \in \text{FTT} : z = u \wedge t \in v\} \end{aligned} \tag{7}$$

To check wether a specific product is working correctly, the test suite is executed on the product (i.e., the resulting set of assets) using the function $working_P$, as shown in (8), where the subscript $_P$ denotes "Product".

$$\begin{aligned} working_P : (\mathcal{P}(\text{TC}), \mathcal{P}(\text{A})) &\rightarrow \{true, false\}, \\ \langle x, y \rangle &\mapsto \forall z \in x : working_S(z, y) \end{aligned} \tag{8}$$

Indeed, the product is working if each test case is executed successfully. That is, the function $working_S$, shown in (9), returns $true$ for the single test case (denoted by the subscript $_S$) executed on the product.

$$\begin{aligned} working_S : (\text{TC}, \mathcal{P}(\text{A})) &\rightarrow \{true, false\}, \\ \langle x, y \rangle &\mapsto \begin{cases} true & \text{if test x succeeds} \\ & \text{with assets y,} \\ false & \text{otherwise} \end{cases} \end{aligned} \tag{9}$$

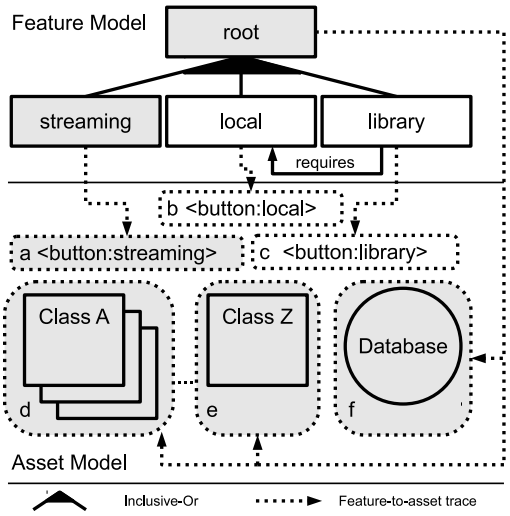

Fig. 1. Overview of a media player product line. Features and assets for a streaming client are marked gray.

Note that feature-to-asset traceability is required in order to obtain an executable configuration from a feature selection on which the required tests for the feature selection are performed. Moreover, obtaining the executable system from feature-to-asset traces keeps the tests for specific features (or combinations thereof) independent of implementation details (i.e., test cases do not have to be adapted but only executed when the implementation of a feature changes).

III. Running Example

To illustrate our approach, we use a simple software product line for media players. The feature model of the media player is depicted in the top part of Fig. 1. A media player can be capable of streaming videos from a server, playing locally stored files, or it can even act as a media library that helps the user organizing his or her media systematically. Formally, the features $\mathrm{F_E}$ are defined in (10) (we use the subscript $_E$ to indicate definitions specific for our running example).

$$\mathrm{F_E} := \{root, streaming, local, library\} \tag{10}$$

The available assets, shown in the lower part of Fig. 1, are formally defined in (11).

$$\mathrm{A_E} := \{a, b, c, d, e, f\} \tag{11}$$

Since at time of initial development of the reference system (i.e., a fully functional media player with all available features) systematic reuse of assets was not a concern for developers, there is little separation of concerns in the source code. For instance, database access is performed directly from individual methods instead of using a common database access object. Thus, finding those pieces of code that actually require a database is not easy. To still make sure that all products of the SPL will work correctly, nearly all assets are simply associated with the root feature, including the database assets, as shown by the dashed arrows in Fig. 1. Formally, this feature-to-asset traceability $\mathrm{FAT_E}$ is defined in (12).

$$\begin{aligned}\mathrm{FAT_E} := \{&\langle root, \{d, e, f\}\rangle, \langle streaming, \{a\}\rangle,\\ &\langle local, \{b\}\rangle, \langle library, \{c\}\rangle\}\end{aligned} \tag{12}$$

The only assets associated with specific features are those that actually display corresponding buttons in the user interface to open a view. Although this example may seem naive, it captures an essential problem that exists when dealing with large and highly complex industrial systems for which a product line should be established.

For testing products, five test cases, as defined in (13), are available.

$$\mathrm{TC_E} := \{tc_1, tc_2, tc_3, tc_4, tc_5\} \tag{13}$$

The feature-to-test traceability FTT_E for the product line is defined in (14).

$$\begin{aligned}\mathrm{FTT}_E := \{&\langle\{root\}, \{tc_1, tc_5\}\rangle, \langle\{streaming\}, \{tc_2\}\rangle,\\ &\langle\{local\}, \{tc_3\}\rangle, \langle\{library\}, \{tc_4\}\rangle\}\end{aligned} \tag{14}$$

Let us now configure a lightweight streaming client that plays videos directly from a server. For this product, the features `root` and `streaming` are selected (as indicated by the gray background of those features in Fig. 1). The corresponding assets for the streaming client are also highlighted in gray in Fig. 1. A formal definition of the specific product $\mathrm{PR_{SC}}$ is shown in (15).

$$\mathrm{PR_{SC}} := \langle\{root, streaming\}, \{a, d, e, f\}\rangle \tag{15}$$

We further assume that $\mathrm{PR_{SC}}$ is the only product configured (i.e., $\mathrm{PR_E} := \{\mathrm{PR_{SC}}\}$). The corresponding test suite $\mathrm{TS_{PR_{SC}}}$ for $\mathrm{PR_{SC}}$ is shown in (16), the testing result $result_{\mathrm{PR_{SC}}}$ is shown in (17).

$$\begin{aligned}\mathrm{TS_{PR_{SC}}} &= testSuite(\{root, streaming\}) \\ &= \{tc_1, tc_2, tc_5\}\end{aligned} \tag{16}$$

$$\begin{aligned}result_{\mathrm{PR_{SC}}} &= working_P(\mathrm{TS_{PR_{SC}}}, \{a, d, e, f\}) \\ &= true\end{aligned} \tag{17}$$

The result shows that the product $\mathrm{PR_{SC}}$ is working and can be delivered to customers.

IV. Optimization of Products

As we have discussed in Section I, there are various reasons why configured products are not optimal. In our running example, the product $\mathrm{PR_{SC}}$ includes various classes and also a full database, thus the risk of, for instance, missing classes is minimized. However, this also means that the product is far from being lightweight as it requires nearly the same amount of disk space as other configurations. Moreover, the product will require an update each time one of the deployed classes is improved – even if the class is not necessary in the product. In safety-critical domains, such *dead assets* may be prohibited at all. Let us now discuss how such issues can be avoided by performing product optimization. We start with a definition of an optimal product.

A product is optimal if it is working and no non-empty subset of the associated assets can be removed without breaking functionality, as shown in (18).

$$\begin{aligned}&optimal : \mathrm{PR} \to \{true, false\}\\ &\langle x, y\rangle \mapsto working_P(testSuite(x), y) \wedge\\ &\nexists z \subset y : working_P(testSuite(x), z)\end{aligned} \tag{18}$$

Indeed, searching for such subsets of a product's assets that can be removed without breaking functionality is possible in theory. However, practically such an approach is not feasible as a product may contain a large number of assets, likely hundreds or thousands, and executing test cases may also require a significant amount of time.

Therefore, we propose a more efficient and practical optimization process that leverages information captured during testing. The optimization process consists of the following two steps: i) obtain the assets in a product that are actually required for providing expected functionality, and ii) identify and remove unnecessary assets from the product. Let us now describe these steps in more detail.

A. Generation of Test-to-Asset Traces

The first step of the optimization process focuses on finding those assets of a product that actually provide the desired functionality. For a given product, a set of test cases required for ensuring correct behavior of the product is available (based on defined feature-to-test traceability). The executable system to be tested is composed by combining the assets associated with the selected features. All required test cases are then executed on the composed system. During testing, all assets that are actually used (e.g., accessed, loaded) are captured and test-to-asset traceability (TAT) is built, as shown in (19).

$$\begin{aligned}\text{TAT} := \{\langle x \in \text{TC}, y \subseteq \text{A}, z \subseteq \text{A}\rangle | \\ \exists \langle a, b\rangle \in PR : x \in testSuite(a) \wedge y = b \wedge z = assets(x, y)\}\end{aligned} \tag{19}$$

The helper function $assets$ is defined as shown in (20) and returns the assets that are used during the execution of a specific test case.

$$\begin{aligned} assets : (\text{TC}, \mathcal{P}(\text{A})) &\rightarrow \mathcal{P}(\text{A}), \\ (x, y) &\mapsto \{z | z \text{ used during } working_S(x, y)\}\end{aligned} \tag{20}$$

Note that the traces are built with data that can be captured during product testing. Thus, there is no need for executing any test cases only for the purpose of trace generation.

For a given product, the used assets can then be derived by the function $usedAssets$, as shown in (21).

$$\begin{aligned} &usedAssets : \text{PR} \rightarrow \mathcal{P}(\text{A}), \\ &\langle x, y\rangle \mapsto \{z \in \text{A} | \exists a \in testSuite(x), \langle b, c, d\rangle \in \text{TAT} : \\ &\qquad a = b \wedge y = c \wedge z \in d\}\end{aligned} \tag{21}$$

B. Removal of Unnecessary Assets

Once the product has been configured and tested, the test-to-asset traces have been established, the second step of the optimization process can be performed. Using the captured test-to-asset traces, traceability between products and their unused assets (PUA) for can be derived, as shown in (22).

$$\text{PUA} := \{\langle \langle x, y\rangle \in \text{PR}, z \subseteq y\rangle | z = y \backslash usedAssets(\langle x, y\rangle)\} \tag{22}$$

Using this information, the set of unnecessary assets can easily be derived for a specific product by the function $unusedAssets$, as shown in (23).

$$\begin{aligned} &unusedAssets : \text{PR} \rightarrow \mathcal{P}(\text{A}), \\ &\langle x, y\rangle \mapsto \{z \in \text{A} | \exists \langle \langle a, b\rangle, c\rangle \in \text{PUA} : x = a \wedge y = b \wedge z \in c\}\end{aligned} \tag{23}$$

Finally, a product can be optimized by removing all unnecessary assets, as shown in (24).

$$\begin{aligned} &optimize : \text{PR} \rightarrow \text{PR}, \\ &\langle x, y\rangle \mapsto \langle x, y \backslash unusedAssets(\langle x, y\rangle)\rangle\end{aligned} \tag{24}$$

Note that removing assets that were not accessed during testing cannot change the overall result of the functionality test. Moreover, all assets remaining in the optimized product are used during testing and thus cannot be removed without affecting functionality. Thus, the optimized product obtained through (24) is optimal with respect to (18). Next, we apply the presented process on our running example.

C. Optimization of Sample Product

After executing the test suite $\text{TS}_{\text{PR}_{\text{SC}}}$, the test-to-asset traceability as shown in (25) is captured.

$$\begin{aligned}\text{TAT}_\text{E} := \{&\langle tc_1, \{a, d, e, f\}, \{e\}\rangle, \\ &\langle tc_2, \{a, d, e, f\}, \{a\}\rangle, \\ &\langle tc_5, \{a, d, e, f\}, \{e\}\rangle\}\end{aligned} \tag{25}$$

When executed, the tests only use the assets a and e. The used assets for the streaming client ($usedAssets_{\text{PR}_{\text{SC}}}$) can then be retrieved as shown in (26).

$$usedAssets_{\text{PR}_{\text{SC}}} = usedAssets(\text{PR}_{\text{SC}}) = \{a, e\} \tag{26}$$

Moreover, the traceability between products and their unused assets can be derived for the sample product line (PUA_E), as shown in (27).

$$\text{PUA}_\text{E} := \{\langle \text{PR}_{\text{SC}}, \{d, f\}\rangle\} \tag{27}$$

The unused assets for the sample product ($unusedAssets_{\text{PR}_{\text{SC}}}$) are then derived as shown in (28).

$$unusedAssets_{\text{PR}_{\text{SC}}} = unusedAssets(\text{PR}_{\text{SC}}) = \{d, f\} \tag{28}$$

Finally, the gathered information is used to derive the optimized (indicated by the superscript $^\text{O}$) product $\text{PR}^\text{O}_{\text{SC}}$, as shown in (29).

$$\begin{aligned}\text{PR}^\text{O}_{\text{SC}} &= optimize(\text{PR}_{\text{SC}}) \\ &= \langle \{root, streaming\}, \{a, e\}\rangle\end{aligned} \tag{29}$$

Note that the optimized product does no longer contain various classes, represented by the asset `d`, and the database, which is represented by asset `f`. Thus, changes of those assets will not trigger unnecessary updates of the streaming client and it requires less disk space.

V. Incremental Handling of SPL Evolution and Further Optimization

So far we have illustrated how individual products can be optimized based on information such as the SPL definition and traces from features to assets and test cases. However, different parts of an SPL typically evolve over time and this evolution also affects optimal products. Specifically, the following evolution scenarios may occur, as depicted on the left-hand side of Fig. 2:

- Feature model change (ΔF)
- Asset model change (ΔA)

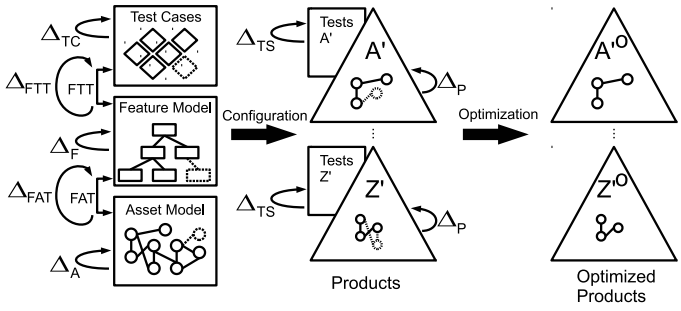

Fig. 2. Points of evolution in an SPL.

- Feature-to-asset traceability change (ΔFAT)
- Test case change (ΔTC)
- Feature-to-test traceability change (ΔFTT)

Those evolution scenarios above may lead to the following changes, as shown in the middle part (Products) of Fig. 2:

- Product configuration change (ΔP)
- Test suite change (ΔTS)

The theory presented in Section II and Section IV allows for handling evolution and deriving the effects on optimal versions of products incrementally. Next, we discuss the effects of different SPL evolution scenarios in detail. We focus on evolution that changes FAT, TC, or FTT because such changes are most complex to process. Note that changes of P, F, A, and TS cause effects that are subsets of the scenarios presented below and are thus not discussed in detail.

A. Feature-to-Asset Traceability Change (ΔFAT)

We start with changes of feature-to-asset traceability. Such changes may occur for various reasons. For instance, existing assets may no longer implement a specific feature because of reductions in customers' requirements and quality expectations. If customers' requirements and quality expectations do increase, however, it may be necessary to add more assets to a feature to reach the expected functionality and level of quality.

1) Addition of Implementation Asset: Let us first consider a case where the asset $a \in \text{A}$ should be added to the feature $f \in \text{F}$ (i.e., traceability between f and a should be established). The set of affected products consists of all products that do include f but do not include other feature to which the asset is already linked. Note that the latter condition is equivalent to the absence of the added asset in the product. Formally, those products can be derived as shown in Eq. (30).

$$\begin{aligned} affProd_{\text{FAT}_\text{A}} : \langle \text{F}, \text{A} \rangle \rightarrow \mathcal{P}(\text{PR}), \\ \langle x, y \rangle \mapsto \{\langle a, b \rangle \in \text{PR} | x \in a \wedge y \notin b\} \end{aligned} \tag{30}$$

For all those products, a re-execution of their complete test suite is necessary as the asset a will be added and may be used in any of the applied test cases.[1] This means that new test-to-asset traces have to be generated, as it is done by the function defined in Eq. (31).

$$\begin{aligned} genTraces_{\text{FAT}_\text{A}} : (\text{F}, \text{A}) \rightarrow \{\langle x \in \text{TC}, y \subseteq \text{A}, z \subseteq y \rangle\}, \\ (x, y) \mapsto \{\langle a, b, c \rangle | \exists \langle d, e \rangle \in affProd_{\text{FAT}_\text{A}}(x, y), \\ tc \in testSuite(d) : \\ a = tc \wedge b = e \cup y \wedge c = assets(tc, e \cup \{y\})\} \end{aligned} \tag{31}$$

As the asset a is added to the affected products, existing test-to-asset traces may become outdated. This is the case if there is no unaffected product for which the trace is still required. The function shown in Eq. (32) can be used to derive those traces.

$$\begin{aligned} outdTraces_{\text{FAT}_\text{A}} : (\text{F}, \text{A}) \rightarrow \mathcal{P}(\text{TAT}), \\ (x, y) \mapsto \{\langle a, b, c \rangle \in \text{TAT} | \nexists \langle d, e \rangle \in \text{PR} \backslash affProd_{\text{FAT}_\text{A}}(x, y) : \\ a \in testSuite(d) \wedge b = e \end{aligned} \tag{32}$$

Based on the generated test-to-asset traces, those assets that are additionally used by a specific product can be derived as shown in Eq. (33).

$$\begin{aligned} addAssets_{\text{FAT}_\text{A}} : (\text{PR}, \text{F}, \text{A}) \rightarrow \mathcal{P}(\text{A}), \\ (\langle x, y \rangle, f, a) \mapsto \{z \in \text{A} | (\exists tc \in testSuite(x), \\ \langle b, c, d \rangle \in genTraces_{\text{FAT}_\text{A}}(f, a) : \\ b = tc \wedge c = y \cup \{a\} \wedge z \in d) \wedge z \notin usedAssets(\langle x, y \rangle)\} \end{aligned} \tag{33}$$

Assets that are no longer used by a specific product can be derived as shown in Eq. (34).

$$\begin{aligned} outdAssets_{\text{FAT}_\text{A}} : (\text{PR}, \text{F}, \text{A}) \rightarrow \mathcal{P}(\text{A}), \\ (\langle x, y \rangle, f, a) \mapsto \{z \in \text{A} | z \in usedAssets(\langle x, y \rangle) \wedge \\ \neg (\exists tc \in testSuite(x), \langle b, c, d \rangle \in genTraces_{\text{FAT}_\text{A}}(f, a) : \\ b = tc \wedge c = y \cup \{a\} \wedge z \in d)\} \end{aligned} \tag{34}$$

The optimal version of a product is affected if additional assets are used by its test suite or previously used assets are no longer required, as shown in Eq. (35).

$$\begin{aligned} optAff_{\text{FAT}_\text{A}} : \langle \text{PR}, \text{F}, \text{A} \rangle \rightarrow \{true, false\}, \\ (\langle x, y \rangle, f, a) \mapsto addAssets_{\text{FAT}_\text{A}}(\langle x, y \rangle, f, a) \neq \emptyset \vee \\ outdAssets_{\text{FAT}_\text{A}}(\langle x, y \rangle, f, a) \neq \emptyset \end{aligned} \tag{35}$$

Only for those products with an affected optimal version, as derived in Eq. (36), the optimization (i.e., the removal of unnecessary assets) must be done again.

$$\begin{aligned} reOptProd_{\text{FAT}_\text{A}} : (\text{F}, \text{A}) \rightarrow \mathcal{P}(\text{PR}), \\ (x, y) \mapsto \{z | z \in affProd_{\text{FAT}_\text{A}}(x, y) \wedge \\ optAff_{\text{FAT}_\text{A}}(z, y)\} \end{aligned} \tag{36}$$

The new optimized products for the addition of a feature-to-asset trace between f and a can be derived incrementally as shown in Eq. (37).

$$\begin{aligned} optimize_{\text{FAT}_\text{A}} : (\text{F}, \text{A}) \rightarrow \langle \langle u, v \rangle \in \text{PR}, \langle x \subseteq \text{F}, y \subseteq \text{A} \rangle \rangle, \\ (f, a) \mapsto \langle \langle u, v \rangle, \langle x, y \rangle \rangle | \exists \langle u, v \rangle \in reOptProd_{\text{FAT}_\text{A}}(f, a) : \\ \langle i, j \rangle = optimize(\langle u, v \rangle) \wedge x = u \wedge \\ y = (j \backslash outdAssets_{\text{FAT}_\text{A}}(\langle u, v \rangle, f, a)) \\ \cup addAssets_{\text{FAT}_\text{A}}(\langle u, v \rangle, f, a) \end{aligned} \tag{37}$$

[1] Note that the number of test cases to re-execute may be reduced by applying optimization techniques [16].

Note that this function returns pairs of the affected products and their updated optimal version.

Finally, the existing test-to-asset traces TAT can be updated to TAT′, as shown in Eq. (38).

$$\text{TAT}' = (\text{TAT} \backslash outdTraces_{\text{FAT}_\text{A}}(f,a)) \cup genTraces_{\text{FAT}_\text{A}}(f,a) \tag{38}$$

Moreover, the set of products can be updated to include the asset a in the affected products, as shown in Eq. (39).

$$\begin{aligned} \text{PR}' := \text{PR} \backslash affProd_{\text{FAT}_\text{A}}(f,a) \cup \{\langle x,y\rangle | \\ \exists \langle i,j\rangle \in affProd_{\text{FAT}_\text{A}}(f,a) : x = i \wedge y = j \cup \{a\}\} \end{aligned} \tag{39}$$

2) Removal of Implementation Asset: Let us now consider the case where the existing trace between a feature $f \in \text{F}$ and an asset $a \in \text{A}$ should be removed (i.e., a is no longer implementing f). In this case, all products that include f are potentially affected. The set of actually affected products contains only those which do not include another feature which is implemented by the asset (i.e., the change in FAT will actually lead to a removal of the asset from the product), as shown in Eq. (40).

$$\begin{aligned} affProd_{\text{FAT}_\text{R}} : \langle \text{F}, \text{A}\rangle \rightarrow \mathcal{P}(\text{PR}), \\ \langle x,y\rangle \mapsto \{\langle a,b\rangle \in \text{PR} | x \in a \wedge \neg(\exists z \in a, \exists \langle u,v\rangle \in \text{FAT} : \\ z \neq x \wedge z = u \wedge b = v)\} \end{aligned} \tag{40}$$

For each of the actually affected products, only the test cases that used the no longer required asset have to be re-executed and new test-to-asset traceability must be generated. Eq. (41) shows how those test cases can be derived for a specific product.

$$\begin{aligned} affTestCases_{\text{FAT}_\text{R}} : \langle \text{PR}, \text{A}\rangle \rightarrow \mathcal{P}(\text{TC}), \\ \langle\langle x,y\rangle, z\rangle \mapsto \{v \in \text{TC} | v \in testSuite(x) \wedge \exists \langle a,b,c\rangle \in \text{TAT} : \\ a = v \wedge b = y \wedge z \in c\} \end{aligned} \tag{41}$$

The new test-to-asset traces can then be derived as shown in Eq. (42).

$$\begin{aligned} genTraces_{\text{FAT}_\text{R}} : (\text{F}, \text{A}) \rightarrow \{\langle a \in \text{TC}, b \subseteq \text{A}, c \subseteq b\rangle\}, \\ (x,y) \mapsto \{\langle a,b,c\rangle | \exists \langle d,e\rangle \in affProd_{\text{FAT}_\text{R}}(x,y), \\ \exists tc \in affTestCases_{\text{FAT}_\text{R}}(\langle d,e\rangle, y) : \\ a = tc, b = e \backslash \{y\} \wedge c = assets(tc,b)\} \end{aligned} \tag{42}$$

For all unaffected test cases, updated test-to-asset traces for the updated product with the removed asset can be derived from existing information without re-execution, as shown in Eq. (43).

$$\begin{aligned} updatedTraces_{\text{FAT}_\text{R}} : \langle \text{PR}, \text{A}\rangle \rightarrow \{\langle a \in \text{TC}, b \subseteq \text{A}, c \subseteq b\rangle\}, \\ \langle\langle x,y\rangle, z\rangle \mapsto \{\langle a,b,c\rangle | \\ a \in testSuite(x) \backslash affTestCases_{\text{FAT}_\text{R}}(\langle x,y\rangle, z) \wedge \\ \exists \langle d,e,f\rangle \in \text{TAT} : a = d \wedge y = e \wedge c = f \wedge b = y \backslash \{z\}\} \end{aligned} \tag{43}$$

The combined set of new traces can be derived as shown in Eq. (44).

$$\begin{aligned} newTraces_{\text{FAT}_\text{R}} : (\text{F}, \text{A}) \rightarrow \{\langle a \in \text{TC}, b \subseteq \text{A}, c \subseteq b\rangle\}, \\ (x,y) \mapsto \{z | z \in genTraces_{\text{FAT}_\text{R}}(x,y) \vee \\ \exists p \in affProd_{\text{FAT}_\text{R}}(x,y) : z \in updatedTraces_{\text{FAT}_\text{R}}(p,y)\} \end{aligned} \tag{44}$$

The outdated test-to-asset traces (i.e., existing traces that are no longer required after a is removed from the affected products) can then be derived as shown in Eq. (45)

$$\begin{aligned} outdTraces_{\text{FAT}_\text{R}} : (\text{F}, \text{A}) \rightarrow \mathcal{P}(\text{TAT}), \\ (x,y) \mapsto \{\langle a,b,c\rangle \in \text{TAT} | \nexists \langle d,e\rangle \in \text{PR} \backslash affProd_{\text{FAT}_\text{R}}(x,y) : \\ a \in testSuite(d), b = e \end{aligned} \tag{45}$$

For each affected product, the assets that are additionally used by the test suite can be derived, as shown in Eq. (46).

$$\begin{aligned} addAssets_{\text{FAT}_\text{R}} : (\text{PR}, \text{F}, \text{A}) \rightarrow \mathcal{P}(\text{A}), \\ (\langle x,y\rangle, f, a) \mapsto \{z \in \text{A} | (\exists tc \in testSuite(x), \\ \langle b,c,d\rangle \in newTraces_{\text{FAT}_\text{R}}(f,a) : \\ b = tc \wedge c = y \backslash \{a\} \wedge z \in d) \wedge z \notin usedAssets(\langle x,y\rangle)\} \end{aligned} \tag{46}$$

Assets that are no longer used by an affected product can be derived as shown in Eq. (47).

$$\begin{aligned} outdAssets_{\text{FAT}_\text{R}} : (\text{PR}, \text{F}, \text{A}) \rightarrow \mathcal{P}(\text{A}), \\ (\langle x,y\rangle, f, a) \mapsto \{z \in \text{A} | z \in usedAssets(\langle x,y\rangle) \wedge \\ \neg(\exists tc \in testSuite(x), \langle b,c,d\rangle \in newTraces_{\text{FAT}_\text{R}}(f,a) : \\ b = tc \wedge c = y \backslash \{a\} \wedge z \in d)\} \end{aligned} \tag{47}$$

The optimal version of the product is affected if either i) additional assets are used by its test suite or ii) previously used assets are no longer required, as shown in Eq. (48).

$$\begin{aligned} optAff_{\text{FAT}_\text{R}} : \langle \text{PR}, \text{F}, \text{A}\rangle \rightarrow \{true, false\}, \\ (\langle x,y\rangle, f, a) \mapsto addAssets_{\text{FAT}_\text{R}}(\langle x,y\rangle, f, a) \neq \emptyset \vee \\ outdAssets_{\text{FAT}_\text{R}}(\langle x,y\rangle, f, a) \neq \emptyset \end{aligned} \tag{48}$$

Only for the set of products with affected optimal version, as derived in Eq. (49), the optimization (i.e., removal of unnecessary assets) must be done again.

$$\begin{aligned} reOptProd_{\text{FAT}_\text{R}} : (\text{F}, \text{A}) \rightarrow \mathcal{P}(\text{PR}), \\ (x,y) \mapsto \{z | z \in affProd_{\text{FAT}_\text{R}}(x,y) \wedge optAff_{\text{FAT}_\text{R}}(z,x,y)\} \end{aligned} \tag{49}$$

For those products, the new optimized product can be derived incrementally as shown in Eq. (50).

$$\begin{aligned} optimize_{\text{FAT}_\text{R}} : (\text{PR}, \text{F}, \text{A}) \rightarrow \langle\langle u,v\rangle \in \text{PR}, \langle x \subseteq \text{F}, y \subseteq \text{A}\rangle\rangle, \\ (f,a) \mapsto \langle\langle u,v\rangle, \langle x,y\rangle\rangle | \exists \langle u,v\rangle \in reOptProd_{\text{FAT}_\text{R}} : \\ \langle i,j\rangle = optimize(\langle u,v\rangle) \wedge x = u \wedge \\ y = (j \backslash outdAssets_{\text{FAT}_\text{R}}(\langle u,v\rangle, f, a)) \\ \cup addAssets_{\text{FAT}_\text{R}}(\langle u,v\rangle, f, a) \end{aligned} \tag{50}$$

Finally, the test-to-asset traces and the affected products can be updated, as shown in Eq. (51) and Eq. (52), respectively.

$$\text{TAT}' = (\text{TAT} \backslash outdTraces_{\text{FAT}_\text{R}}(f,a)) \cup newTraces_{\text{FAT}_\text{R}}(f,a) \tag{51}$$

$$\begin{aligned} \text{PR}' := \text{PR} \backslash affProd_{\text{FAT}_\text{R}}(f,a) \cup \{\langle x,y\rangle | \\ \exists \langle i,j\rangle \in affProd_{\text{FAT}_\text{R}}(f,a) : x = i \wedge y = j \backslash \{a\}\} \end{aligned} \tag{52}$$

B. Test Case Change (ΔTC)

Let us now consider an evolution scenario in which a test case $tc \in \text{TC}$ is adapted as a consequence of, for instance, increased quality requirements. In this scenario, all products that include the changed test case tc in their test suite are affected, as shown in Eq. (53).

$$\begin{aligned} affected_{\text{TC}} : \text{TC} \rightarrow \mathcal{P}(\text{PR}), \\ x \mapsto \{\langle y, z\rangle | x \in testSuite(y)\} \end{aligned} \tag{53}$$

From existing test-to-asset traces, those traces that were previously generated for tc become outdated. They can be derived as shown in Eq. (54).

$$\begin{aligned} oldTraces_{\text{TC}} : \text{TC} \rightarrow \mathcal{P}(\text{TAT}), \\ tc \mapsto \{\langle x, y, z\rangle \in \text{TAT} | x = tc\} \end{aligned} \tag{54}$$

For each affected product $pr \in affected_{\text{TC}}(tc)$, tc must be re-executed. This generates a new set of test-to-asset traces, as shown in Eq. (55).

$$\begin{aligned} newTraces_{\text{TC}} : \text{TC} \rightarrow \{\langle x \in \text{TC}, y \subseteq \text{A}, z \subseteq y\rangle\}, \\ tc \mapsto \{\langle x, y, z\rangle | x = tc \wedge \exists \langle a, b\rangle \in affected_{\text{TC}}(tc) : \\ y = b \wedge z = assets(tc, y)\} \end{aligned} \tag{55}$$

For each of the affected products $pr \in affected_{\text{TC}}(tc)$, the additional assets that are used by the updated test case and also those assets that are no longer used can be derived. If the test case uses assets that it did not use before the update, the additional assets must be added to the optimal version of the product if no other test case in the test suite also uses those assets. Similarly, if assets that were used before the update are no longer used afterwards, they must be removed from the optimal product if they are not also used by another test case. The functions for deriving the sets of additionally and no longer used assets are shown in Eq. (56) and Eq. (56), respectively.

$$\begin{aligned} addUsedAssets_{\text{TC}} : (\text{TC}, \text{PR}) \rightarrow \mathcal{P}(\text{A}), \\ (tc, \langle x, y\rangle) \mapsto \{z | z \in y \wedge \exists \langle a, b, c\rangle \in newTraces_{\text{TC}}(tc), \\ \langle d, e, f\rangle \in oldTraces_{\text{TC}}(tc) : \\ a = d \wedge b = e \wedge b = y \wedge z \in c \wedge z \notin f\} \end{aligned} \tag{56}$$

$$\begin{aligned} outdAssets_{\text{TC}} : (\text{TC}, \text{PR}) \rightarrow \mathcal{P}(\text{A}), \\ (tc, \langle x, y\rangle) \mapsto \{z | z \in y \wedge (\exists \langle a, b, c\rangle \in oldTraces_{\text{TC}}(tc), \\ \langle d, e, f\rangle \in newTraces_{\text{TC}}(tc) : \\ a = d \wedge b = e \wedge b = y \wedge z \in c \wedge z \notin f) \wedge \\ \neg(\exists \langle g, h, i\rangle \in \text{TAT}, j \in testSuite(x) : \\ j \neq tc \wedge g = j \wedge h = y \wedge z \in i)\} \end{aligned} \tag{57}$$

The new optimized products for all affected products can then be derived by adding the set of newly used assets and removing the set of no longer used assets, as shown in Eq. (58).

$$\begin{aligned} optimize_{\text{TC}} : \text{TC} \rightarrow \{\langle \langle x, y\rangle \in \text{PR}, \langle a \subseteq \text{F}, b \subseteq \text{A}\rangle\rangle\}, \\ (tc) \mapsto \langle \langle x, y\rangle, \langle a, b\rangle\rangle | \exists \langle x, y\rangle \in affected_{\text{TC}} : \\ \langle u, v\rangle = optimize(\langle x, y\rangle) \wedge a = x \wedge \\ b = v \backslash outdAssets_{\text{TC}}(tc, \langle x, y\rangle)) \\ \cup addUsedAssets_{\text{TC}}(tc, \langle x, y\rangle)) \end{aligned} \tag{58}$$

Finally, the test-to-asset traceability TAT can be updated to TAT′ as shown in Eq. (59).

$$\text{TAT}' := (\text{TAT} \backslash oldTraces_{\text{TC}}(tc)) \cup newTraces_{\text{TC}}(tc) \tag{59}$$

C. Feature-to-Test Traceability Change (ΔFTT)

Consumer expectations and market demands change over time. Thus, it is possible that certain features become more crucial for consumer decisions and a product that includes a specific feature is expected to include more actual functionality than before. We have already discussed changing consumer expectations may lead to updates of test cases in order to, for example, make a test more rigourous with respect to certain quality measures. However, often the consumer's expectations do not only change in terms of expected quality but also in terms of functionality. While for the latter is would be possible to adapt existing test cases, adding additional test cases for checking the newly expected functionality is often preferred as it helps keeping individual test cases separated and maintainable. In principle, changed customer expectations may also make existing test cases obsolete. In both cases, traceability between features and test cases must be adapted.

1) Addition of Trace: Let us now discuss how the addition of a new test case $tc \in \text{TC}$ that checks functionality of the feature $f \in \text{F}$ is handled (i.e., a feature-to-test trace between f and tc should be added). In this scenario, only those products are affected that do include f and do not include another feature that is already linked to the test case tc, as shown in Eq. (60).

$$\begin{aligned} affProd_{\text{FTT}_\text{A}} : (\mathcal{P}(\text{F}), \text{TC}) \rightarrow \mathcal{P}(\text{PR}), \\ (x, y) \mapsto \{\langle a, b\rangle \in \text{PR} | y \notin testSuite(a) \wedge x \subseteq a\} \end{aligned} \tag{60}$$

For those affected products, the test case must be executed and new test-to-asset traces are generated, as shown in Eq. (61).

$$\begin{aligned} newTraces_{\text{FTT}_\text{A}} : (\mathcal{P}(\text{F}), \text{TC}) \rightarrow \{\langle a \in \text{TC}, b \subseteq \text{A}, c \subseteq \text{A}\rangle\}, \\ (x, y) \mapsto \{\langle a, b, c\rangle | a = y \wedge \exists \langle u, v\rangle \in affProd_{\text{FTT}_\text{A}}(x, y) : \\ b = v \wedge c = assets(y, v)\} \end{aligned} \tag{61}$$

If, for a specific affected product, tc uses assets that are not yet used by other test cases, those assets must be added to the optimal product, similar to test case changes that lead to more assets being used. The assets that are used only by the newly required test case can be derived as shown in Eq. (62).

$$\begin{aligned} newAssets_{\text{FTT}_\text{A}} : (\mathcal{P}(\text{F}), \text{TC}, \text{PR}) \rightarrow \mathcal{P}(\text{A}), \\ (w, x, \langle y, z\rangle) \mapsto \{a | \exists \langle b, c, d\rangle \in newTraces_{\text{FTT}_\text{A}}(w, x) : \\ b = x \wedge c = z \wedge a \in d \wedge a \notin optimize(\langle y, z\rangle)\} \end{aligned} \tag{62}$$

The new optimized products for a desired addition of a new feature-to-test trace can be derived incrementally as shown in Eq. (63)

$$\begin{aligned} optimize_{\text{FTT}_\text{A}} : (\mathcal{P}(\text{F}), \text{TC}) \rightarrow \{\langle \langle c, d\rangle \in \text{PR}, \langle a \subseteq \text{F}, b \subseteq \text{A}\rangle\rangle\}, \\ (x, y) \mapsto \{\langle \langle c, d\rangle, \langle a, b\rangle\rangle | \exists \langle c, d\rangle \in affProd_{\text{FTT}_\text{A}}(x, y) : \\ a = c \wedge b = optimize(c, d) \cup newAssets_{\text{FTT}_\text{A}}(x, y, \langle c, d\rangle)\} \end{aligned} \tag{63}$$

Finally, the test-to-asset traces must be updated and the trace between f and tc must be added to construct the updated

feature-to-test traceability FTT′, as shown in Eq. (64) and Eq. (65), respectively.

$$\text{TAT}' := \text{TAT} \cup newTraces_{\text{FTT}_\text{A}}(x, y) \tag{64}$$

$$\text{FTT}' := \text{FTT} \cup \{\langle\{f\}, \{tc\}\rangle\} \tag{65}$$

2) Removal of Trace: Now, let us discuss how the desired removal of an existing feature-to-test trace between the feature $f \in \text{F}$ and the test case $tc \in \text{TC}$ is handled. If the trace between f and tc should be removed, then those products are affected that include tc in their test suite and do not include other features that also require the test case, as shown in Eq. (66).

$$\begin{aligned} affProd_{\text{FTT}_\text{R}} : (\mathcal{P}(\text{F}), \text{TC}) \rightarrow \mathcal{P}(\text{PR}), \\ (x, y) \mapsto \{\langle a, b\rangle \in \text{PR} | x \subseteq a \wedge \\ \nexists z \subseteq a, \langle c, d\rangle \in \text{FTT} : z \neq x \wedge z = c \wedge y \in d\} \end{aligned} \tag{66}$$

The test-to-asset traces for affected products and the no longer required test case tc can be retrieved using the function shown in Eq. (67).

$$\begin{aligned} affTraces_{\text{FTT}_\text{R}} : (\mathcal{P}(\text{F}), \text{TC}) \rightarrow \mathcal{P}(\text{TAT}), \\ (x, y) \mapsto \{\langle a, b, c\rangle \in \text{TAT} | a = y \wedge \\ \exists \langle d, e\rangle \in affProd_{\text{FTT}_\text{R}}(x, y) : e = b\} \end{aligned} \tag{67}$$

As the test case tc is removed from test suites of affected products, test-to-asset traces may become obsolete. Those outdated traces can be derived as shown in Eq. (68).

$$\begin{aligned} outdTraces_{\text{FTT}_\text{R}} : (\mathcal{P}(\text{F}), \text{TC}) \rightarrow \mathcal{P}(\text{TAT}), \\ (x, y) \mapsto \{\langle a, b, c\rangle \in affTraces_{\text{FTT}_\text{R}}(x, y) | \\ \nexists \langle f, g\rangle \in \text{PR} \backslash affProd_{\text{FTT}_\text{R}} : a \in testSuite(f) \wedge g = b\} \end{aligned} \tag{68}$$

For an affected product, the assets that are used by tc but not by any other test cases in the test suite can be removed from the optimized version of the product; Eq. (69) shows how those assets are derived.

$$\begin{aligned} outdAssets_{\text{FTT}_\text{R}} : (\mathcal{P}(\text{F}), \text{TC}, \text{PR}) \rightarrow \mathcal{P}(\text{A}), \\ (x, y, \langle a, b\rangle) \mapsto \{c \in \text{A} | c \in b \wedge \\ (\exists \langle d, e, f\rangle \in affTraces_{\text{FTT}_\text{R}}(x, y) : \\ d = y \wedge e = b \wedge c \in f) \wedge \\ (\nexists \langle g, h, i\rangle \in \text{TAT}, v \in testSuite(a) : \\ v \neq y \wedge v = g \wedge h = b \wedge c \in i)\} \end{aligned} \tag{69}$$

The optimized versions of affected products for the intended change can then be derived incrementally as shown in Eq. (70).

$$\begin{aligned} optimize_{\text{FTT}_\text{R}} : (\mathcal{P}(\text{F}), \text{TC}) \rightarrow \{\langle\langle c, d\rangle \in \text{PR}, \langle a \subseteq \text{F}, b \subseteq \text{A}\rangle\rangle\}, \\ (x, y) \mapsto \{\langle\langle c, d\rangle, \langle a, b\rangle\rangle | \exists \langle c, d\rangle \in affProd_{\text{FTT}_\text{R}}(x, y) : \\ a = c \wedge b = optimize(c, d) \backslash outdAssets_{\text{FTT}_\text{R}}(x, y, \langle c, d\rangle)\} \end{aligned} \tag{70}$$

Of course, the test-to-asset and feature-to-test traces must be updated, as shown in Eq. (71) and Eq. (72), respectively.

$$\text{TAT}' := \text{TAT} \backslash outdTraces_{\text{FTT}_\text{R}}(x, y) \tag{71}$$

$$\text{FTT}' := \text{FTT} \backslash \{\langle\{f\}, \{tc\}\rangle\} \tag{72}$$

VI. Validation

To demonstrate the feasibility of the approach and assess its efficiency, we implemented a prototype tool for SPL modeling and product optimization.[2] Please note that neither composing products from individual assets (e.g., [11], [17]) nor capturing traces during the execution of systems (e.g., [18], [19]) is a technical challenge. Thus, we focused on validating whether our approach is capable of performing incremental re-optimization of products efficiently in case of SPL evolution.

A. Sample SPLs

For the tests, we randomly generated SPL with the following characteristics: 25–500 features, 100–10,000 assets, 25–1,000 test cases, 25–1,000 product configurations. To each feature, a randomly selected set of implementing assets (with a size between 1% and 5% of total assets) and a randomly selected set of test cases (also with a size between 1% and 5% of total test cases) were assigned. Each generated product consisted of a randomly selected set of features (containing between 5% and 10% of all features). Each product's test suite and unoptimized assets were derived from the feature selection. For each test case execution, a random subset of the tested product's unoptimized assets was defined to be used. In total, 4,509 SPLs were generated.

B. Efficiency Tests

To asses the efficiency of our approach, the tests described below were performed on each of the sample SPLs. The benchmarks were run on an Intel Core i5-650 machine with 8GB of memory running Windows 7 Professional. For every test, the median processing time for 100 executions was used for our statistics. Note that the captured processing times do not include times for test case execution. This is because those test cases would have been executed for regression testing anyway [20]. Thus, the performed tests assess the additional effort required by our approach. Moreover, we only captured changes that actually required a re-optimization of at least one product. Five different tests were defined, one for each of the evolution scenarios: ΔFAT (add (I) and remove (II)), ΔFTT (add (III) and remove (IV)), and ΔTC (V) in Section V. For each of the sample SPLs, each test was executed with every possible combination of feature and asset or test case (for (I) and (III)), each existing trace (for (II) and (IV)), or each existing test case (for (V)).

C. Results

To analyze the efficiency and scalability of our approach, we regressed the required processing time on the most important variables (i.e., number of features, assets, test cases, products, affected products, and re-optimized products). The results are depicted in Table I. Note that the regression model also allowed for quadratic growth. The results indicate that for the squared values of SPL characteristic variables (e.g., number of features) and the number of affected products the estimated coefficients are either negative or below 0.005. Therefore, we expect that increasing values for those variables do not cause exponential growth in processing times.

[2] Available at http://www.sea.jku.at/tools/pospl.

TABLE I. REGRESSION RESULTS FOR EVOLUTION SCENARIOS.

Time(ms)	Add FAT (I)	Remove FAT (II)	Add FTT (III)	Remove FTT (IV)	Change TC (V)
Features	0.919***	0.701***	0.0215***	0.114***	0.149***
	(0.00327)	(0.00249)	(0.000488)	(0.0104)	(0.0203)
Features^2	-0.000767***	-0.000572***	-0.0000324***	-0.000175**	0.00143***
	(0.00000362)	(0.00000288)	(0.00000283)	(0.0000629)	(0.000117)
Assets	-0.00660***	-0.00886**	-0.0000198	-0.000998*	-0.00105
	(0.000834)	(0.00277)	(0.0000174)	(0.000428)	(0.000960)
Assets^2	0.00000199***	0.00000171***	2.55e-08***	0.000000166***	0.000000318***
	(7.68e-08)	(0.000000252)	(1.60e-09)	(3.93e-08)	(8.80e-08)
Testcases	0.698***	0.527***	0.00645***	0.0792***	0.370***
	(0.00376)	(0.00235)	(0.000236)	(0.00931)	(0.0121)
Testcases^2	0.000662***	0.000137***	-0.00000819***	-0.000234***	-0.000885***
	(0.00000428)	(0.00000238)	(0.000000734)	(0.0000403)	(0.0000426)
Products	-1.035***	-1.071***	-0.00456***	-0.0102***	0.0568***
	(0.00520)	(0.00403)	(0.000107)	(0.00234)	(0.00246)
Products^2	0.000571***	0.000503***	0.00000555***	0.00000510**	-0.0000299***
	(0.00000415)	(0.00000318)	(8.30e-08)	(0.00000171)	(0.00000227)
Affected	28.80***	19.37***	0.0602***	0.366***	0.486***
	(0.277)	(0.122)	(0.00169)	(0.0348)	(0.0293)
Affected^2	-0.668***	-0.194***	-0.0000304	-0.000454	0.00464***
	(0.0134)	(0.00437)	(0.0000186)	(0.000355)	(0.000290)
Re-optimized	-7.377***	4.285***	0.121***	0.605***	0
	(0.265)	(0.113)	(0.00175)	(0.0352)	(N/A)
Re-optimized^2	0.498***	0.00305	0.00269***	-0.00699***	0
	(0.0133)	(0.00435)	(0.0000534)	(0.000907)	(N/A)
Constant	-187.0***	-148.1***	-2.265***	-14.04***	-52.59***
	(0.939)	(2.552)	(0.0278)	(0.693)	(1.227)
Observations	1170383	815781	546816	15760	24388
R^2	0.672	0.746	0.589	0.551	0.522

Standard errors in parentheses
* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

For Test (I), the regression shows that the number of affected products and the number of re-optimizations are the most important factors for determining processing time. The mean processing time observed for Test (I) was 127ms for changes that required a re-optimization of 12.3 products on average. The average number of features for this test was 134.5. For Test (II), the results are similar to those of Test (I). However, the number of affected and the especially the number of re-optimized products do have a significantly smaller effect on processing times. For Test (III) and Test (IV), we can see that the most important variable is the number of re-optimized products. For Test (IV), the second most important variable is the number of affected products. The average times observed for Test (III) and Test (IV) are 1.9ms and 8.3ms, respectively. The mean number of re-optimized products per change was 5.5 for both tests. The estimation for Test (V) shows that the time for processing a test case update depends primarily on the number of affected products. The mean processing time was 20.6ms and 7.2 products were re-optimized on average.

D. Threats to Validity

The use of random SPLs in our validation may be seen as a threat to validity. Clearly, those SPLs do not necessarily represent SPLs found in practice. However, characteristics of real SPLs are also highly heterogeneous due to factors such as the domain, employed technology, or also coding style. Therefore, we chose to use random SPLs to cover a broad variety of possible SPLs instead of relying on few real-world examples that may not be representative.

Our approach also requires the traces between features and test cases to be correct in order to apply the correct set of tests and thus find the correct set of actually used assets. However, note that these test cases are functionality tests for the resulting product. Therefore, we believe that this is a justifiable assumption since the connection between features and the functionality their selection should add to the product is usually well documented. Moreover, research has shown that test case selection based on feature selections is feasible [21].

VII. RELATED WORK

Software product lines are an active field of research. We now discuss approaches and developments closest to ours.

A relevant approach in terms of SPL consistency checking is *Safe Composition* [2], which would typically infer all products that can be configured for a given feature model are valid with respect to the asset model (i.e., that all products are working). Our approach focuses on practical aspects and performs optimization only for configured products that are working. By combining our approach with safe composition, only products that are expected to work (with respect to known dependencies between assets) can be configured and have to be empirically tested and then optimized.

Various techniques have been proposed to construct SPLs, including feature models and traceability, from already existing products. Rubin and Chechik [22] proposed approaches for constructing a feature-oriented SPL through refactoring and finding suitable feature-to-asset traces. Linsbauer et al. [15] use existing products and retrieving feature-to-asset traces. These approaches take the asset combinations of the existing products as given. With our optimization approach, those products can be optimized so that only relevant traceability is generated.

Regarding SPL optimization, Guo et al. [23] presented an approach for optimizing feature selections with respect to resource constraints. They rely on feature models that are annotated with resource information and optimize feature selections

so that certain resource constraints are met by the selection. While they derive a feature combination that provides a good match for given resource constraints, our approach focuses on optimizing the resource consumption for a given selection.

Cordy et al. [24] investigated the effects of product line evolution. Based on feature transition systems, different categories of features, and behavioral analysis they determine which products have to be model checked after feature model evolution. Sabouri et al. [25] proposed an approach for handling the evolution of feature models. They use state-spaces for possible products and support the reuse of consistency results for the existing state space when checking the consistency of the evolved product line using a *Spin* model checker. One of biggest differences between our optimization approach and those approaches is that we do not use traditional model-checking techniques for finding possible areas of optimization in an SPL. The theory presented in this paper in principle enable the incremental determination of affected products and test cases to be re-executed for checking whether those products still work. However, instead of relying on abstract formalizations, we leverage data gained during testing of the actual product as it would be delivered to the customer. In doing so, we avoid the necessity of translating the SPL definition to formal concepts, which is typically non-trivial as dependencies between complex assets must be defined precisely in order to get valid reasoning results from model checkers. Furthermore, our approach supports the evolution of all parts of an SPL. Lochau et al. [26] presented an incremental approach for managing test suites for product line regression testing. They use deltas to determine which test cases become required or obsolete as the consequence of evolution. Their approach enables an efficient updating of feature-to-asset traces as it is required in our approach. Neves et al. [14] proposed templates for various SPL evolution scenarios that guide designers during the process. While templates provide valuable guidance, our approach instantly checks the evolved product line's actual validity – also in cases where no templates are available for the evolution scenario.

VIII. Conclusions

In the paper we presented the basic formal concepts of a product optimization approach for SPLs. Our approach leverages functionality tests for finding and eliminating unnecessary assets from individual products. We have shown that the presented concepts support automatic and incremental handling of SPL evolution. Performance tests showed that the approach scales and provides quick information about the effects of SPL evolution on products. The optimization results can also be used for further optimization of variability and for future work we plan to investigate those possibilities.

Acknowledgments

The research was funded by the Austrian Science Fund (FWF): P21321-N15, and FWF Lise-Meitner Fellowship M1421-N15.

References

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software product line engineering - foundations, principles, and techniques*. Springer, 2005.

[2] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook, "Safe composition of product lines," in *GPCE*, pp. 95–104, 2007.

[3] K. Czarnecki and U. W. Eisenecker, *Generative programming - methods, tools and applications*. Addison-Wesley, 2000.

[4] N. I. Altintas, S. Cetin, A. H. Dogru, and H. Oguztüzün, "Modeling product line software assets using domain-specific kits," *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1376–1402, 2012.

[5] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, and K. Pohl, "Variability issues in software product lines," in *PFE*, pp. 13–21, 2001.

[6] K. Pohl and A. Metzger, "Software product line testing," *Commun. ACM*, vol. 49, no. 12, pp. 78–81, 2006.

[7] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho, "Extracting and evolving code in product lines with aspect-oriented programming," *T. Aspect-Oriented Software Development*, vol. 4, pp. 117–142, 2007.

[8] M. Revelle and D. Poshyvanyk, "An exploratory study on assessing feature location techniques," in *ICPC*, pp. 218–222, 2009.

[9] B. S. Andersen and G. Romanski, "Verification of safety-critical software," *Commun. ACM*, vol. 54, no. 10, pp. 52–57, 2011.

[10] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," *Journal of Systems and Software*, vol. 74, no. 2, pp. 173–194, 2005.

[11] S. Apel, C. Kästner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *ICSE*, pp. 221–231, 2009.

[12] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE*, pp. 422–437, 2005.

[13] C. W. Krueger, "Easing the transition to software mass customization," in *PFE*, pp. 282–293, 2001.

[14] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza, and P. Borba, "Investigating the safe evolution of software product lines," in *GPCE*, pp. 33–42, 2011.

[15] L. Linsbauer, R. Lopez-Herrejon, and A. Egyed, "Recovering traceability between features and code in product variants," in *SPLC*, 2013.

[16] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test., Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, 2012.

[17] S. Boxleitner, S. Apel, and C. Kästner, "Language-independent quantification and weaving for feature composition," in *Software Composition*, pp. 45–54, 2009.

[18] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: a framework for application performance monitoring and dynamic software analysis," in *ICPE*, pp. 247–248, 2012.

[19] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok, "Software monitoring with controllable overhead," *STTT*, vol. 14, no. 3, pp. 327–347, 2012.

[20] W. Heider, R. Rabiser, P. Grünbacher, and D. Lettner, "Using regression testing to analyze the impact of changes to variability models on products," in *SPLC*, pp. 196–205, 2012.

[21] S. Wang, A. Gotlieb, S. Ali, and M. Liaaen, "Automated test case selection using feature model: An industrial case study," in *MoDELS*, pp. 237–253, 2013.

[22] J. Rubin and M. Chechik, "Combining related products into product lines," in *FASE*, pp. 285–300, 2012.

[23] J. Guo, J. White, G. Wang, J. Li, and Y. Wang, "A genetic algorithm for optimized feature selection with resource constraints in software product lines," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2208–2221, 2011.

[24] M. Cordy, A. Classen, P.-Y. Schobbens, P. Heymans, and A. Legay, "Managing evolution in software product lines: a model-checking perspective," in *VaMoS*, pp. 183–191, 2012.

[25] H. Sabouri and R. Khosravi, "Efficient verification of evolving software product lines," in *FSEN*, pp. 351–358, 2011.

[26] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity, "Incremental model-based testing of delta-oriented software product lines," in *TAP*, pp. 67–82, 2012.