Don’t Worry About it: Managing Variability On-The-Fly

Sebastian Krieter
Harz University of Applied Sciences
Wernigerode, Germany
Otto-von-Guericke-University
Magdeburg, Germany
skrieter@hs-harz.de

Jacob Krüger
Harz University of Applied Sciences
Wernigerode, Germany
Otto-von-Guericke-University
Magdeburg, Germany
jkrueger@hs-harz.de

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
tleich@hs-harz.de

ABSTRACT

Software-product-line engineering (SPLE) has become a widely adopted concept to implement reusable source code. However, instead of using SPLE from the beginning (the proactive approach), a software product line (SPL) is often only introduced after a set of similar systems is already developed (the extractive approach). This can lead to additional costs, new bugs introduced by refactoring, and an overall inconsistent SPL. In particular, inconsistencies between the variability implemented in the source code and the one represented in a variability model can become a major problem. To address this issue, we propose the concept of *variability management derivation*: We aim to (semi-)automatically model features and their dependencies while developers implement variable source code to facilitate the initial development, reusability, and later maintainability of SPLs, utilizing the reactive approach. In this paper, we demonstrate our concept by means of preprocessors. However, we claim that it can be adapted for other SPLE implementation techniques to facilitate SPL development.

CCS CONCEPTS

• **Software and its engineering → Software product lines**;

KEYWORDS

Software product line, adoption strategy, reactive development, variability model

ACM Reference Format:
Sebastian Krieter, Jacob Krüger, and Thomas Leich. 2018. Don’t Worry About it: Managing Variability On-The-Fly. In *VAMOS 2018: 12th International Workshop on Variability Modelling of Software-Intensive Systems, February 7–9, 2018, Madrid, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3168365.3170426

1 INTRODUCTION

Software reuse is one of the most important concepts in software engineering to reduce development costs [4, 10, 14, 43]. Employing software reuse results in a set of similar systems that comprise shared and unique code alike. For example, copying source code and adapting parts of it is one of the most simplistic reuse approaches [19, 38]. This approach is known as *copy-and-paste* for smaller code parts and *clone-and-own* for entire systems [18, 38]. Each derived clone includes modified behavior referred to as *variability*. Therefore, the concept of variability is closely connected to software reuse.

Listing 1: Example of C preprocessor code.

```
1 void login(char* name, char* pw) {
2 # ifdef HTTPS
3   login_https(server, name, pw);
4 # endif
5 # ifdef SSH
6   login_ssh(server, name, pw);
7 # endif
8 }
```

SPLE has been proposed as a more systematic reuse approach than clone-and-own [4, 13, 14]. Functionality (described as *features*) is implemented just once in domain artifacts, which are then reused to derive customized product variants. In practice, mainly preprocessors are used to implement SPLs [4, 21, 34]. Here, a single code base exists in which variable source code is marked with annotations and is removed before compilation – if the corresponding feature is not part of the configuration. We show an example using the C preprocessor (CPP) [25] in Listing 1. For the CPP, `#ifdef` directives mark the beginning and `#endif` directives the end of a variable code part. A feature’s name is defined in the condition that follows an `#ifdef`, for instance `HTTPS` (cf. Line 2 in Listing 1). While our proposed concept is not limited to preprocessors, we will focus on them in this work to provide a consistent example. Thus, if we refer to introducing variability with a preprocessors, we refer to any situation in which a feature is made reusable also, for example, via cloning or components. However, further adaptations to our concept are necessary in this context, for example, a strong integration of the underlying platform in the case of components.

When developers start implementing a system, they may not think about its reusability. In particular, this applies if they are unfamiliar with SPLE, develop completely new products, or are working in a less software-intensive domain. However, at some point in time, variability in the system can become necessary and developers start to implement variable code. For example in Listing 1, the functionality of logging via `HTTPS` may have been present from the beginning, but at some point new requirements led to the implementation of an `SSH` feature to improve the security in some systems. Although the C preprocessor – along with many other variability mechanisms – benefits from using a feature model, it does not explicitly require one, which means that developers are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VAMOS 2018, February 7–9, 2018, Madrid, Spain
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5398-4/18/02…$15.00
https://doi.org/10.1145/3168365.3170426

not forced to model this newly introduced variability. Thus, over time the knowledge about variable and reusable source code will fade and needs to be recovered [20, 31], hampering the maintenance and comprehensibility of the code.

To address this problem, we propose the concept of *variability management derivation* to support developers in managing reusable code. The idea of this concept is to semi-automatically generate and refine variability models [7, 11, 15] whenever developers add variability to their code. This exceeds current capabilities of SPLE, its tools, and corresponding implementation techniques that either do not use models (e.g., preprocessors) or require a full model defined by the developer (e.g., feature-oriented programming [37]). Currently, we envision to extend a development environment to support developers by semi-automatically defining dependencies of new features as soon as they are implemented. However, even further improvements can be achieved by a stronger integration of this concept into respective programming languages, resulting in more and more variability managing being handled automatically. Thus, later on *variability management derivation* may also comprise other aspects of reuse and variability, for instance, deriving and customizing test cases to configurations.

2 SOFTWARE PRODUCT LINE ADOPTION

In this section, we describe the three adoption strategies for SPLs introduced by Krueger [28]. For each, we motivate how *variability management derivation* can support the adoption strategy. To do so, we rely on the terms *problem space*, referring to an abstracted view (e.g., based on variability models) on the domain, and *solution space*, referring to the actually implemented domain artifacts [4, 14].

2.1 Proactive Adoption

Proactive development is often considered to be the ideal adoption strategy for SPLE: While requiring the highest upfront investment, the break-even point can be reached faster [12, 30, 40]. For proactive adoption, the SPL is developed from scratch and fully designed before its implementation. Thus, the complete problem space is scoped and then a corresponding solution space is derived.

In this context, *variability management derivation* can help during the development, as even in a modeled problem space some new features may be added during the implementation or maintenance phase. Here, our concept supports the developer in managing, documenting, and mapping newly added functionality. Furthermore, different code constructs, for example the nesting of preprocessor directives, could contradict or imply different variability than defined in the problem space [33, 35, 42]. With our concept, we can already then identify such cases and call for the developer's attention to update the model or source code to ensure consistency.

2.2 Extractive Adoption

Far more common in practice than the proactive approach is the extractive adoption strategy [4, 6, 34]. In this case, a set of often cloned systems, comprising commonalities and variabilities, exist. These are than migrated into an SPL. Thus, different solution spaces preexist and have to be joined as well as mapped into a single problem space [31].

Here, *variability management derivation* can help during the extraction process to ensure that the resulting SPL is consistent and to prevent faults in the mappings. Thus, we facilitate and support the developer while refactoring the cloned systems, reducing the costs and risks of extractions [12, 30]. Furthermore, we can support the step of locating features and deriving a variability model, which can only be semi-automated [9, 23, 31, 32].

2.3 Reactive Adoption

Finally, the reactive approach refers to situations in which a single system is developed and later extended towards an SPL. Thus, variability may already be planned from the beginning, integrating the proactive approach, or just be added whenever it seems appropriate. For instance, in preprocessor-based systems, features can simply be added and made configurable by including corresponding annotations. Still, these changes in the solution space are often not mapped to the problem space, resulting in outdated documentations, variability models, and mappings – ultimately requiring to locate features again during maintenance.

We propose *variability management derivation* especially for this adoption strategy to facilitate the introduction of variability on-the-fly. Thus, whenever a developer adds a feature, for example by including preprocessor annotations, we immediately analyze the code, trying to automatically identify its dependencies and map it to the variability model. Still, this is hardly possible in all situations, wherefore we propose to raise the developers' awareness for the newly introduced variability at this point. Our concept can help to facilitate the management of reuse and variability while implementing code by proposing changes to or even automatically updating the variability model.

We remark that we base our following concept on *feature models* as they are commonly used in practice and academia [7, 11]. Nonetheless, other variability models can also be used or are even necessary for specific implementation techniques. For example, for delta-oriented programming the corresponding delta models have to be used [39].

3 PROBLEM STATEMENT

The variability of an SPL is provided by both, feature modeling (problem space) and variation points in the source code (solution space). While the feature model explicitly defines valid configurations, the source code implicitly defines valid products due to its data and control dependencies.

We call an SPL *consistent*, if the variability that is intended by the developers is reflected in the source code and enforced by the feature model. Otherwise we call the SPL *inconsistent*. More precisely, for an SPL to be consistent, we require certain properties of feature model and source code. The source code should facilitate developers to understand the implemented behavior – which represents one of the main activities of developers [46, 47] – and, thus, comprise the following properties:

- comprehensibility,
- meaningful variability, and
- feature traceability.

Consequentially, to be easily comprehensible, the source code needs to have an appropriate annotation structure (i.e., nesting) and must

reflect the variability in the feature model rather than declaring all features as optional. Furthermore, each code block must be mapped to at least one feature in the feature model to achieve reasonable feature traceability. Simultaneously, the feature model should forbid configurations that could result in:

- duplicated products,
- syntactical errors, and
- semantical faults.

As not all variability concepts of a feature model can be perfectly represented by all implementation techniques, our definition of consistency serves more as a guideline than a measurable property. Still, we find it suitable as foundation of our concept of *variability management derivation*.

The goal of this concept is to support developers in implementing such a consistent SPL, keeping feature model and source code aligned to ensure that the implemented variability represents the intended one. This alignment is necessary to fulfill the aforementioned properties that support developers differently in their tasks. On the one hand, properly representing variability in the code facilitates feature location, traceability, and code comprehension. On the other hand, a consistent feature model provides a simpler overview of the SPL's structure, enforces dependencies that cannot completely be represented in the code, is necessary to prevent that invalid configurations are instantiated, and enables model-based analysis. Thus, consistently representing variability throughout the SPL (i.e., in both code and feature model) provides several benefits.

4 RUNNING EXAMPLE

Throughout the paper, we use an example SPL of a cloud-storage client that contains seven features: `Client`, `Crypto`, `Encryption`, `Signing`, `Login`, `HTTPS`, and `SSH`. We represent the feature model of this example SPL as a feature diagram in Figure 1. The features `Encryption` and `Signing` are in an *or*-relationship, which means that if their parent feature `Crypto` is selected, at least one of both must be selected, too. Similarly, the features `HTTPS` and `SSH` are in an *alternative*-relationship, meaning that exactly one of them must be selected if their parent `Login` is selected. This is always the case, as `Login` is a mandatory child of the root feature `Client`.

To exemplify potential inconsistencies between this model and its implementation according to our definition and motivate *variability management derivation*, assume the following: Considering our preprocessor example in Listing 1 and only the annotations for `HTTPS` and `SSH`, we can get the impression that both features are optional and independent from one another. However, when we take a deeper look at the source code, we notice that configurations in which both or none of the features are selected do not make sense, as there is either no login or the login is executed twice, respectively. Thus, we would model an alternative between both features, as we show in the feature model in Figure 1, and generate variable code, which we display in Listing 2. This clarifies the developer's intentions for others and, at the same time, enforces the intended variability via the feature model by prohibiting meaningless product configurations.

With our concept of *variability management derivation*, we strive to resolve such inconsistencies already during development, to improve the management of SPLs and avoid maintenance problems,

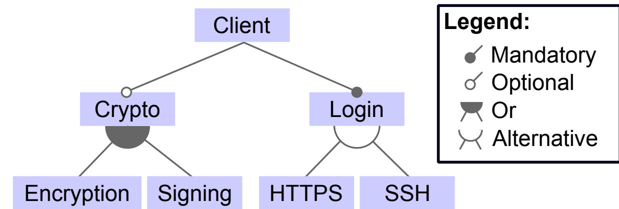

Figure 1: Feature model of the example cloud SPL.

Listing 2: Login method with generated annotations.

```
1  #ifdef Login
2  void login(char* name, char* pw) {
3  # ifdef HTTPS
4    login_https(server, name, pw);
5  # elif SSH
6    login_ssh(server, name, pw);
7  # else
8  #   error "Invalid configuration!"
9  # endif
10 }
11 #endif
```

as well as unnecessary or faulty analyses. One of our methods is to reflect the variability of the feature model within the source code, which introduces redundancies (cf. Line 8 at Listing 2). While this may seem counterintuitive at first, these additional information can help developers to properly understand the variability at certain parts of the code and facilitate feature location [20]. Using tool support, we aim to keep the accruing implementation overhead for the developers to a minimum.

5 VARIABILITY MANAGEMENT DERIVATION

In order to resolve inconsistencies between the feature model and the source code, we first need to consider the feature modeling process. Feature modeling consists of two main tasks: Identifying distinct features and specifying their dependencies [4]. When using feature diagrams as feature model representations, feature dependencies can be displayed by both, the tree structure and additional cross-tree constraints. Thus, we have to consider three activities with our approach to promote a consistent feature model:

(1) identifying features,
(2) modeling the tree structure, and
(3) adding cross-tree constraints.

With *variability management derivation*, we intend to support the developer in these activities by (partly) deriving the feature model from variation points within the source code. Our main goal is to automatize each activity as far as possible. Furthermore, if automation is not possible or not reasonable, we still want to guide developers by providing meaningful suggestions and calling for their attention. In this section, we describe each activity and how we intend to support it in more detail.

5.1 Identifying Features

One of the biggest challenges is to identify features within the source code. In particular, feature identification and location is hard to automate, as a tool doesn't know a developer's intentions [9, 23].

Identifying Variant Features. We plan to provide tool support for the reactive approach in the following way: Alternatively to adding a preprocessor annotation, developers can simply mark a code block and assign a feature via a command (e.g., context menu, hot key, ...). The tool then creates a new feature in the feature model and generates the corresponding source code annotations. With this, we achieve two goals: Assigning a feature to a code block (potentially by creating a new one in the feature model) and enabling variability within the code, while requiring equal or even less effort from the developer. In contrast, if a developer manually defines a variation point in the code, the feature name may be automatically derived (e.g., based on the preprocessor directive) and mapped to the model. For both cases, the developer may need to define dependencies of the newly introduced feature, which we aim to support with automatic suggestions.

We illustrate our described technique in the example given in Listing 3. Here, we can imagine that a developer is implementing an `upload` method, which uploads some packages to the cloud server. At the beginning, the developer is just implementing the non-highlighted parts without considering any variability. To this end, the feature model is almost empty, containing only the root feature `Client`. Later on, the developer is adding the variable code highlighted in yellow. Each of the three highlighted lines comprises a different variation point. The `crypto` library added in Line 1 is used in Line 3 and 4 to sign or encrypt the given package, respectively. Instead of adding annotations manually, the developer marks each line and specifies a feature. Then, the tool automatically generates annotations in the code and adds the corresponding feature to the feature model. We depict the final source code in Listing 4 and the resulting feature model on the left side of Figure 2.

Identifying Core Features. Another issue concerning feature identification is the mapping of source code to core features (i.e., features that are present in every valid product [5]). While using our proposed concept, developers can easily assign features to certain variable parts of the source code and identify all variant features (i.e., features that are present in only some valid products [5]). Still, the question arises how to induce the developer to also model core features. Arguably, this may be unnecessary if we only aim to consistently model variability. However for maintenance purposes and later adaptations (e.g., making a core feature optional), we argue that these tasks can be facilitated if all features are mapped to the code and their dependencies are defined. Then, it is not necessary during an update to recover feature locations and investigate its dependencies, which can be costly tasks [20, 26, 30, 48].

With our tool, we plan to automatically provide suggestions to the developers to define features for new computational units, such as, classes and methods. Thus, if performed consistently, all parts of the code are mapped to a certain feature. The default suggestions could be to map core functionality to the root feature of the feature model or, if existing, the feature of a larger encapsulating unit, for instance of a class including a considered method. Alternatively, our tools may propose to create a new feature for classes, for example based on the class name. The developer can agree with a suggestion or add the unit to another feature. For example, in Listing 4, we may suggest to map the non-variable parts of the method to the root feature `Client` (i.e., Line 4, 11, and 12). If accepted, corresponding annotations are added, as we show in Listing 5. This way, traceability of any kind of feature is supported, facilitating the comprehension of a system. However, we may not use preprocessor directives for this purpose, but lightweight documentation annotations, for example as proposed by Ji et al. [20]—thus, avoiding additional configuration options that would increase complexity.

Listing 3: Upload method with highlighted variable code.

```
1  #include <crypto_lib.h>
2  void upload(char* package) {
3    package = crypto_sign(package);
4    package = crypto_encrypt(package);
5    upload_to_server(server, package);
6  }
```

Listing 4: Upload method with generated annotations.

```
1  #ifdef Crypto
2  # include <crypto_lib.h>
3  #endif
4  void upload(char* package) {
5  # ifdef Signing
6    package = crypto_sign(package);
7  # endif
8  # ifdef Encryption
9    package = crypto_encrypt(package);
10 # endif
11   upload_to_server(server, package);
12 }
```

Listing 5: Upload method with generated annotations.

```
1  #ifdef Client
2  #ifdef Crypto
3  # include <crypto_lib.h>
4  #endif
5  void upload(char* package) {
6  # ifdef Crypto
7  # ifdef Signing
8    package = crypto_sign(package);
9  # endif
10 # ifdef Encryption
11   package = crypto_encrypt(package);
12 # endif
13 # endif
14   upload_to_server(server, package);
15 }
16 # endif
```

5.2 Modeling Feature Dependencies

After identifying features within the source code, we need to model their dependencies.

Defining Alternatives. One possibility to define a feature relationship is by introducing an alternative variation point in the code and reflecting it in the model. By now, we only consider variation points that add functionality to the source code. However, we can also use variation points that provide alternative implementations of the same functionality. Similar to additions, a developer can mark the relevant code blocks and define them as alternative. Another possible method is that the developer marks just one code block and then implements the alternative code block(s) on-the-fly. Again, the

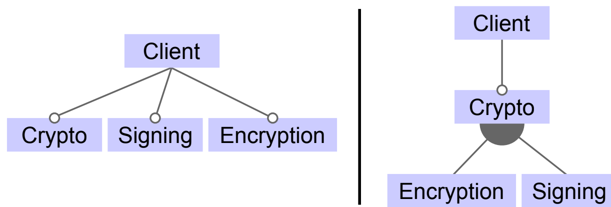

Figure 2: Feature model of the example cloud SPL.

Listing 6: Login method with highlighted variable code.

```
1  void login(char* name, char* pw) {
2    login_https(server, name, pw);
3    login_ssh(server, name, pw);
4  }
```

tool would generate annotations in the source code and restructure the feature model accordingly.

When we consider our first example of the method `login`, we can show an application of this technique. In Listing 6, we depict the source code without annotations, again highlighting the variable parts in yellow. The developer would mark both lines, assign a feature to each of them and define them as alternative implementations. Subsequently, the tool would generate annotations and add the new features and their dependencies to the feature model. When creating an alternative group, the tool requires a corresponding parent feature. In this example, we assume that the developer creates a new feature `Login`. Analogous to Listing 5, we can wrap the annotations of the child features. However, as the entire method body consists only of variable code, we can extend the annotation to encapsulate the whole login method. We show the resulting code in Listing 2 and the final feature model in Figure 1. If an alternative to an already existing feature is implemented, we can ideally automatically reflect this in the model, as we know the parent and child features. Still, our tool will report an inconsistency if other features are defined below the parent, which requires either to assign the alternative group to another parent or assign the existing features to the implemented alternatives.

Analyzing the Source Code. Regarding the feature tree structure, we aim to utilize source code analysis, for example for preprocessor-based variability [35], to extract feature dependencies. Additional metrics, variability mechanics (e.g., **`#else`** directives, nesting, or boolean operators for preprocessors), and naming conventions can be used to refine the results or identify further dependencies. Still, automatically extracting all dependencies is hardly possible and user support will be necessary.

In our previous example in Listing 4, we identified the three involved features `Crypto`, `Encryption`, and `Signing`. However, currently all of them are optional, which is represented on the left side of Figure 2 but differs from the intended variability that we can see in the source code. Both, `Encryption` as well as `Signing` require the inclusion of the `crypto` library by the feature `Crypto`. Thus, there is a clear relationship $(Encryption \lor Signing) \rightarrow Crypto$. We can simply model this relationship within the feature tree by making `Crypto` the parent of `Encryption` and `Signing`.

In addition, we have the relation $Crypto \rightarrow (Encryption \lor Signing)$, as it does not make sense to include the `Crypto` library if neither signing nor encryption is intended. Thus, the tool support will suggest a corresponding restructuring of the diagram, which we show on the right side of Figure 2. Additionally, the tool will update the source code by generating an additional annotation, such that the relationship of the three features is also clear from the code. In Listing 5, we can see the new annotation at Line 6, which is encapsulating the variation points for signing and encryption.

5.3 Adding Cross-Tree Constraints

For cross-tree constraints, the same automatic analyses as for the previous steps can be helpful. Additional metrics can help to identify dependencies that are not represented in the tree structure, for example:

- Local control flow dependencies (within a method/class)
- Global control flow dependencies (entire system)
- Data flow analysis
- Architectural dependencies (e.g., packages, namespaces)

However, cross-tree constraints are even more challenging to automatically identify than the tree structure. Thus, these constraints heavily rely on the developer to be manually specified. To support this, our approach will suggest dependencies based on static analysis of the source code and verify newly defined dependencies. Similar, to alternatives and additions, we can also reflect cross-tree constraints to the source code by using expression in the annotations.

5.4 Special Cases

As we already indicated, there exist some special cases of adding variability that we need to consider within our concept. In this section, we briefly describe some of these cases and how we address them using *variability management derivation*.

Distributed Alternative Code. In contrast to our small example in Listing 2, there might also be alternative code blocks that need to be implemented at different locations within the code (e.g, in another class or method). Here, we cannot use a simple **`#ifdef`**-**`#else`** structure. However, instead of using **`#ifdef`** with a single directive, we can introduce a more complex expression in the respective annotations that also reflects the intended variability, for instance, `#if defined SSH && !defined HTTPS`.

Manual Annotating. If the developer chooses to just add annotations without the mechanism we provide, we can automatically identify that variability is introduced due to the preprocessor directives. We then can also identify the feature's name by analyzing the macro definition and even match whether it is already defined in the model or not. In the first case, the tool can check for inconsistencies between the intended and implemented variability using existing approaches [22, 35], asking the developer to resolve issues. For the second case, we aim to propose the developer to include the feature in the model and potentially propose a suitable position, for example, depending on the nesting of annotations. Even if the developer chooses not to include the feature, we can still add it as an optional one to at least represent it in the model. Thus, we aim to prevent inconsistencies between model and source code.

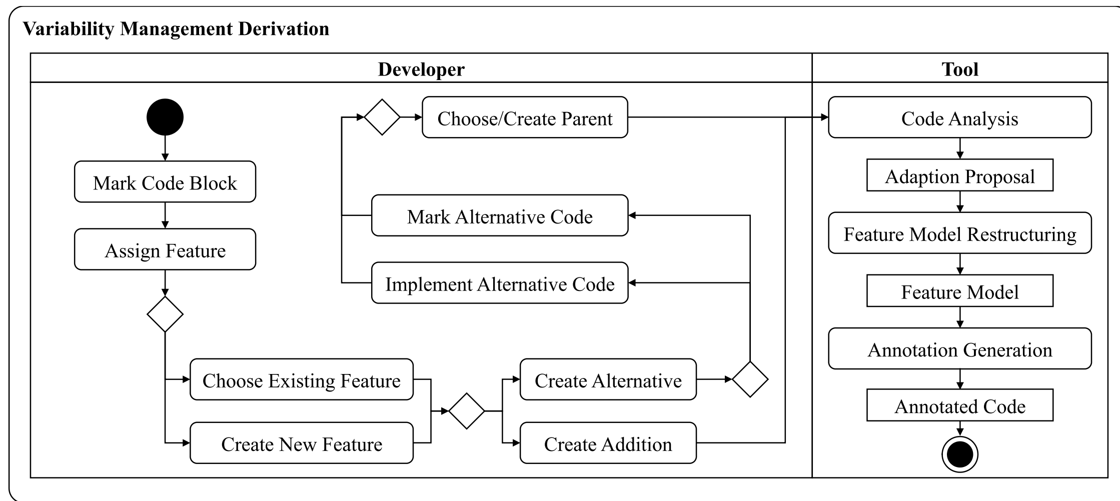

Figure 3: Envisioned workflow of *variability management derivation* (activity diagram).

Changing Presence Condition. Changing the presence conditions of a variable code block can lead to broken dependencies. We aim to address this issues within our tool as follows: When an analysis detects a problem introduced by the modified variable code, we either warn the developer or, if possible, generate an appropriate constraint that fixes the problem by reducing the problem space. Again, if the developer manually changes such a presence condition without using the provided mechanism, we derive and propose solutions for the identified issues to the developer.

5.5 Envisioned Work-Flow

Summing up the previously described activities, we now describe our envisioned work-flow of *variability management derivation*. We depict this work-flow – considering that the developer uses the functionalities of our tool and does not solely rely on its on-the-fly analysis – as an activity diagram in Figure 3. On the left side, we show all activities and decisions that require input from a developer. Analogous, on the right side, we show the activities that are executed by the development environment.

The developers start by *marking a code block* that they consider as variable. Next, they *assign a feature* to the code block and can either *choose an existing feature* from the feature model or provide a name to *add a new, optional feature* to the feature model. Afterwards, the developers have to choose whether the variable code block is an *addition* to the remaining code or part of an *alternative* implementation of a feature. In case of an addition, the developers are effectively done, though they could specify additional dependencies. Otherwise, they have to specify the alternative implementation by either *marking an existing code block* or by *implementing a new one* in the editor. In addition, they have to *define a parent feature* for the alternative group in the feature model. Just as previously, they can either choose an existing feature or create a new one.

After the developers provided their input, the remaining activities are executed by the tool to refine feature model and source code. First, the tool performs a *static code analysis*, considering data and control flow, dependencies, and variable code structures. Based on this analysis, the tool provides a list of proposals for adapting the feature model and source code. If there is more than one proposal, the developer can choose which one the tool should apply. Second, using the results of the source code analysis, the tool *restructures the feature model* accordingly, arranging the features in groups or adding necessary cross-tree constraints. Finally, the tool *generates annotations* in the source code. Each step of the implementation and analysis work-flow interacts with the developer, showing in-between results and issues that call for the developers' attention.

Using the described work-flow, the developer creates a consistent product line and benefits from its feature traceability and comprehensible code structure (cf. Section 3). It also forces developers to reason about their variability decisions, which helps to derive a feature model that represents the developers intentions. While this work-flow may seem to introduce some overhead, in general, we aim to reduce the implementation effort for developers as the tool support helps them to write annotations, adapt the feature model, and find feature interactions. The automatic derivation of structures facilitates development while the included analyses facilitate maintenance by ensuring consistency.

5.6 Implementation Concept

In the following, we describe our initial thoughts on a potential implementation of *variability management derivation*. Rather than implementing a standalone tool, we plan to integrate our approach into FeatureIDE [45], which is an extensible open-source framework for feature-oriented software development written in Java. By extending FeatureIDE, we benefit from its already implemented functionality, including feature modeling, source code navigation and highlighting, and variability analyses.

In order to minimize the overhead introduced by *variability management derivation*, we intend to employ incremental execution of included analyses wherever possible. That is, whenever a new piece of variability is introduced, rather than performing a complete analysis of the entire code, we build on previous results and only analyze code parts that are affected by the most recent change.

6 RELATED WORK

While, in our paper, we mainly aim to support the reactive approach of SPLE, we are also considering the extractive approach and corresponding analyses, which have been extensively analyzed. For example, reverse engineering feature models has been investigated by several authors, relying on different input artifacts, such as source code, propositional formulas, or product descriptions [1, 16, 41]. Not all of the existing approaches can be applied in our concept, due to some artifacts potentially missing. Nonetheless, other approaches can be fully adopted to support automation and we can derive additional ideas from all of these works for our incremental analyses in the reactive approach.

Other works consider type-checking of certain SPLs implementation techniques [17, 22, 44]. As these works aim to ensure type safety, they have to perform static analysis of the systems [24]. We can utilize similar ideas to ensure consistency and type safety in SPLs developed with our concept. For this, we can adopt these approaches, depending on the underlying implementation technique, and use them in the incremental analysis.

There exist some integrated development environments for SPLE, such as FeatureIDE [45], Gears [29], or Pure::Variants [8], that support automation of different steps. They can identify inconsistencies in the variability model, for example, dead features or redundant dependencies. However, this is only done when the model is already fully designed. Thus, errors and inconsistencies can only be exploited and repaired during the testing phase, when derived variants contain syntax errors or behave faulty. In contrast, our main goal is to support developers in preventing inconsistencies from the beginning, while they develop their SPL. Still, we aim to rely on FeatureIDE, implementing our concept as one of its extensions.

Besides fully-fledged tools, some implementation techniques for SPLE, such as feature-oriented programming [37], force the developer to design a variability model. However, this does not account for far more used techniques that employ, for example, preprocessors, plug-ins, or components [4, 34]. In addition, even when a variability model is required, its consistency (especially with the source code) is not validated by the implementation technique. Thus, we support any SPLE approach by checking for consistency during development and improving readability of the source code. Due to the enforced model and underlying mechanisms, additional analysis may be available to further automate our approach.

Some works address analysis and testing of inconsistencies in the variability model as well as between model and source code [2, 3, 27, 33, 36]. Still, most of these works are performed on already developed SPLs and, thus, require a complete variability model for checking consistency. We can utilize such approaches by adopting them to our work-flow and incremental on-the-fly modeling of variability. As a result, we complement these works by adopting them for a use case that can already ensure consistency during development, instead of checking consistency afterwards.

7 CONCLUSION

In this paper, we presented our vision of a tool to support especially the reactive approach towards SPLE. We aim to help developers implementing their intended variability in the source code and map them in a corresponding feature model. We discussed the activities necessary to define variability in the code, to design a feature model, and indicated how these tasks can be facilitated by appropriate tool support. Furthermore, we claim that by using our approach, developers can benefit from several advantages: The source code does reflect the intended variability of the developer, increasing its comprehensibility and, by this, decreasing the chance of introducing bugs when modifying or maintaining the code. Each line in the source code is mapped to at least one feature to improve traceability. The feature model does correspond to the developers' intentions, avoiding modeling faults and unwanted product variants.

In future work, we plan to implement our described approach and evaluate its usefulness in user studies. Here, we aim to compare development efforts but also maintainability and comprehensibility of the resulting SPLs. Furthermore, we want to incorporate and improve already known techniques for code analysis and automated feature modeling. An interesting question is, whether different implementation techniques are easier to integrate and for which our approach may facilitate SPL development more than for others.

ACKNOWLEDGMENTS

This research is supported by DFG grants LE 3382/2-1 and LE 3382/3, and Volkswagen Financial Services AG.

REFERENCES

[1] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. 2012. On Extracting Feature Models from Product Descriptions. In *International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. ACM, 45–54.

[2] Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Mutation Operators for Preprocessor-Based Variability. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 81–88.

[3] Mustafa Al-Hajjaji, Jacob Krüger, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. Efficient Mutation Testing in Configurable Systems. In *International Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, 2–8.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[6] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 1–8.

[7] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.

[8] Danilo Beuche. 2012. Modeling and Building Software Product Lines with Pure::Variants. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 255–255.

[9] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1994. Program Understanding and the Concept Assignment Problem. *Commun. ACM* 37, 5 (1994), 72–82.

[10] Barry W. Boehm, Chris Abts, and Sunita Chulani. 2000. Software Development Cost Estimation Approaches - A Survey. *Annals of Software Engineering* 10, 1 (2000), 177–205.

[11] Lianping Chen and Muhammad Ali Babar. 2011. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology* 53, 4 (2011), 344–362.

[12] Paul C. Clements and Charles W. Krueger. 2002. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software* 19, 4 (2002), 28–30.

[13] Paul C. Clements and Linda M. Northrop. 2002. *Software Product Lines*. Addison-Wesley.

[14] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

[15] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.

[16] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *International Software Product Line Conference (SPLC)*. IEEE, 23–34.

[17] Benjamin Delaware, William R Cook, and Don Batory. 2009. Fitting The Pieces Together: A Machine-Checked Model of Safe Composition. In *Joint Meeting of The European Software Engineering Conference and The ACM SIGSOFT Symposium on The foundations of Software Engineering (ESEC/FSE)*. ACM, 243–252.

[18] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.

[19] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 391–400.

[20] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 61–70.

[21] Christian Kästner and Sven Apel. 2008. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*. University of Passau, 35–40.

[22] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-based Product Lines. *ACM Transactions on Software Engineering and Methodology* 21, 3 (2012), 14:1–14:39.

[23] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features. *IEEE Transactions on Software Engineering* 40, 1 (2014), 67–82.

[24] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 25–32.

[25] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language*. Prentice Hall.

[26] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971–987.

[27] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. In *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 132–143.

[28] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering (PFE)*. Springer, 282–293.

[29] Charles W. Krueger. 2007. BigLever Software Gears and the 3-Tiered SPL Methodology. In *ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 844–845.

[30] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 354–361.

[31] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 65–72.

[32] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. Composing Annotations Without Regret? Practical Experiences using FeatureC. *Software: Practice and Experience* (2017).

[33] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating Consistency Between a Feature Model and Its Implementation. In *International Conference on Software Reuse (ICSR)*. Springer, 1–16.

[34] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *European Conference on Object-Oriented Programming (ECOOP)*, Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 495–518.

[35] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

[36] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts. *Empirical Software Engineering* 21, 4 (2016), 1744–1793.

[37] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 419–443.

[38] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing Forked Product Variants. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 156–160.

[39] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. Springer, 77–91.

[40] Klaus Schmid and Martin Verlage. 2002. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software* 19, 4 (2002), 50–57.

[41] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *International Conference on Software Engineering (ICSE)*. IEEE, 461–470.

[42] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2011. Efficient Extraction and Analysis of Preprocessor-Based Variability. *ACM SIGPLAN Notices* 46, 2 (2011), 33–42.

[43] Thomas A. Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 494–497.

[44] Sahil Thaker, Don Batory, David Kitchin, and William Cook. 2007. Safe Composition of Product Lines. In *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 95–104.

[45] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (2014), 70–85.

[46] Rebecca Tiarks. 2011. What Maintenance Programmers Really Do: An Observational Study. In *Workshop Software Reengineering (WSR)*. 36–37.

[47] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. 1997. Program Understanding Behaviour During Enhancement of Large-Scale Software. *Journal of Software Maintenance: Research and Practice* 9, 5 (1997), 299–327.

[48] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *Journal of Software: Evolution and Process* 25, 11 (2013), 1193–1224.