Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines

Reimar Schröter
University of Magdeburg
Germany

Norbert Siegmund
University of Passau
Germany

Thomas Thüm,
Gunter Saake
University of Magdeburg
Germany

ABSTRACT

Despite the wide use of software product lines, their implementation and evolution is a challenging task. When implementing a feature, a developer has to know which code fragments of other (already implemented) features are accessible in each program variant in which the feature is included. Especially for composition-based implementation techniques, in which the code is implemented in separated modules, it is an exhausting and error-prone task to find safely accessible code fragments of other modules. State-of-the-art tool support, such as product-line type checkers, detect errors a posteriori (i.e., during compilation), but fails to prevent errors during the implementation. To overcome this problem, we propose *feature-context interfaces*, which provide a modular and non-variable programming interface to the variable source code of a product line. These interfaces ease changes, extensions, and the maintainability of product lines. To demonstrate applicability, we implemented a content assist and an outline view in Eclipse based on feature-context interfaces. We evaluate the potential of our method by analyzing the number of potential type errors we prevent compared to state-of-the-art techniques.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*

General Terms

Design, Reliability

Keywords

Software product lines, syntactic interface, modularity

SPLC '14, September 15 - 19 2014, Florence, Italy

http://dx.doi.org/10.1145/2648511.2648522

1. INTRODUCTION

Software product-line (SPL) engineering aims at the development of similar programs based on a common code base [8]. *Features* describe the commonalities and variabilities of different program variants [10]. To generate a tailored program, called *variant*, stakeholders select features, satisfying their requirements. Selected features are mapped to their corresponding implementation units, which are then composed and compiled to obtain customized program variants. The benefits of reduced time-to-market and high reusability led to the development of an increasing number of SPLs in practice, such as HP printer driver, Siemens medical care SPL, the Linux kernel, and Nokia's SPLs [24].

Time-to-market is one of the main benefits of SPLs that is strongly influenced by the development process of the product line itself. An incorrect implementation of a feature can increase development time resulting in higher costs. Several techniques exist to identify an incorrect feature implementation, such as product-line type checkers [2, 15]. However, these techniques identify errors only *after* the implementation of a code artifact, but fail to *prevent* them, again, leading to an increased development time. Moreover, there is no composition-based technique supporting programmers in identifying safely accessible code fragments during the implementation. In detail, on the one hand, there is no overview about accessible code that would prevent compile-time errors. On the other hand, the lack of not knowing which code fragments are safely accessible can lead to unexploited reuse opportunities that hinder the maintenance and the development of new features.

Especially the development of composition-based SPLs, in which developers implement features locally into separate code modules (e.g., components, aspects, or feature modules), challenges the identification of safely accessible code. Let us consider a small example in which we add a new feature to an existing SPL. A developer implementing the new feature f wants to use a class `C` only introduced by feature g. If, on the one hand, g is not necessarily part of every variant that includes the new feature f, a dangling reference to `C` will occur in some variants. On the other hand, if all variants containing f will indeed contain g, not using `C` could be a missed opportunity for reuse. Beside the extension of a product line by feature f, we need to know which code fragment is safe to use for changes in feature implementations or other maintenance tasks of SPLs. With the increasing trend of reusing SPLs itself (i.e., in a *multi software product line (MPL)* [20]), the problem becomes even worse. A developer has additionally to know the dependencies between

the involved SPLs to identify accessible code artifacts.

In general, how can the developer know which methods exist and are safely callable from a certain feature? The developer needs to consider the domain dependencies among features in the SPL (e.g., one feature requires another feature) and has to infer from these dependencies all accessible code fragments. With an increasing number of features and their dependencies, this approach quickly becomes infeasible. To the best of our knowledge, there is no approach that supports the developer and presents a modular and non-variable view on safely reusable code fragments for a currently implemented module.

We propose an interface depending on the current *feature context*. The feature context represents the feature in which the developer is working to extend, maintain, or change the SPL. Based on this context, we compute a non-variable interface, called *feature-context interface*, to the remaining code of the SPL that is safely accessible. By using members of the interface only, we are able to prevent compile-time errors before the compilation and we are aware of reuse opportunities.

We demonstrate the applicability of feature-context interfaces by implementing tool support in FeatureIDE [22], an Eclipse extension for the development of SPLs. In detail, we realize an auto-completion mechanism and an outline view, making feature-context interfaces available for a large community. The current implementation supports software product lines written in Java based on FeatureHouse [3]. Our general approach supports a large range of composition-based implementation techniques and programming languages.

To evaluate the potential of feature-context interfaces, we compute the variable interface of an SPL. The variable interface contains the union of all members defined in all code modules and is used as baseline for comparisons. Our evaluation of eight SPLs indicates that feature-context interfaces reduce the number of members compared to the variable interface, by 39 %, on average. Furthermore, we found that we can reduce up to 28 % of potential type errors compared to a state-of-the-art development. Moreover, some state-of-the-art approaches hide, on average, 51% of members relative to the variable interface that are accessible in the feature-context interface hindering the reusability of code fragments during the development of new features.

In summary, we make the following contributions:

- We introduce feature-context interfaces as a non-variable view to the variable program interface of a composition-based SPL.
- We implemented auto-completion support for code and an outline view in FeatureIDE based on feature-context interfaces.
- We compare feature-context interfaces regarding error prevention to state-of-the-art approaches, such as APIs of program variants and the variable SPL interface.

2. PRELIMINARIES

Next, we give some background on feature modeling and composition-based implementation mechanisms that are relevant for feature-context interfaces.

2.1 Feature Models

A feature model specifies combinations of features that give rise to valid variants [10, 1]. A feature diagram is a graphical representation of a feature model in form of a tree, in which each child feature requires the presence of its parent feature. In Figure 1, we show the feature diagram of a navigation SPL (ignore source code for now). The root feature *Navigation* is present in all variants, similar to all mandatory child features (*Graph*, *Wgt*, and *Gtp*), which must be included in a variant iff the parent (i.e., feature *Navigation*) is included. In contrast, feature *Connected* is an optional feature that can be included if the parent is included. Furthermore, feature diagrams allow us to define *or groups* and *alternative groups*. If the parent feature of an *or group* is part of a variant, at least one grouped feature has to be included, too. Features *Car* and *Pedestrian* are defined in an alternative group (represented as an arc) meaning that exactly one of these features must be included if the parent is included. A feature diagram can have also cross-tree constraints in the form of propositional formulas, or exclude ($\leftrightarrow$) and require ($\rightarrow$) edges. The *Navigation* SPL defines two cross-tree constraints ($Car \rightarrow Weighted$ and $Car \rightarrow Directed$), which enforce that each variant with feature *Car* also includes features *Weighted* and *Directed*.

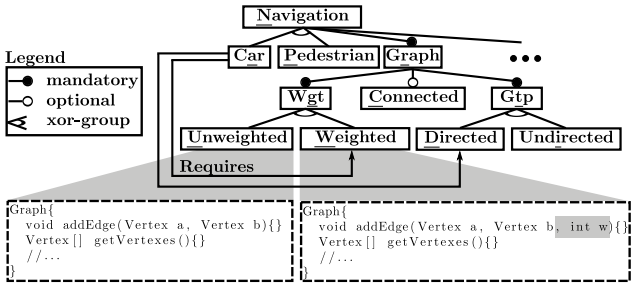

Figure 1: Different program interfaces through feature selection.

A propositional formula is an alternative feature-model representation which we can also use to describe the dependencies in the navigation SPL:

$$n \wedge r \wedge g \wedge t \wedge (a \leftrightarrow \neg p) \\ \wedge (u \leftrightarrow \neg w) \wedge (d \leftrightarrow \neg i) \wedge (a \rightarrow w \wedge d)$$

Each feature is represented by a unique variable (we use the underlined characters of Figure 1 for brevity). Based on this representation, a satisfiability solver allows us to check whether a given feature selection is valid [7]. In this paper, we use the representation as propositional formula to retrieve feature-context interfaces.

2.2 Composition-Based Implementation

In feature-oriented SPLs, software is decomposed in units of functionality that map to a set of features F [1]. Whereas composition-based techniques modularize a feature's implementation (e.g., using feature modules), annotation-based approaches use annotations to map code to features (e.g., #ifdef directives). We focus on composition-based techniques, especially on feature-oriented programming [6, 17], in which the code of a feature $f_i \in F$ is implemented by a dedicated *feature module* denoted as $impl(f_i)$. Feature modules that correspond to selected features are composed to generate a variant v:

$$impl(v) = impl(f_1) \bullet ... \bullet impl(f_n) \ with \ f_1, ..., f_n \in F$$

where $\bullet : I \times I \rightarrow I$ is the composition operator that is defined over the set I of implementation units and $f_1, ..., f_n$ are the selected features of the variant v [5]. In detail, each feature module consists of a set of classes (including methods, fields, etc.), which are composed via superimposition with equally named classes of other selected feature modules, and recursively with equally named members in classes having the same type. The resulting classes contain the superimposed members of all selected implementation units.

For example, each feature *Graph*, *Wgt*, ..., *Connected* $\in F$ of the navigation SPL has its own feature module *impl*(*Graph*), *impl*(*Wgt*), ..., *impl*(*Connected*). In Figure 2, we depict the composition of feature modules *impl*(*Graph*) and *impl*(*Unweighted*). Here, the class `Graph` is defined in both features, but with different methods and fields. Class `Vertex` is defined only in feature *Graph*, but used in both feature modules. The superimposition of both feature modules result in two classes `Graph` and `Vertex` containing the members of both modules.

3. PROBLEM STATEMENT

Implementing SPLs is challenging. Depending on the currently implemented feature, the code fragments (i.e., the set of classes, methods, and fields) that are safely accessible differ caused by domain constraints that require or forbid a different set of features. We denote the feature context (i.e., the currently implemented feature module) *FC* as one feature in F, such that $FC \in F$.

Example. In Figure 1, we show the feature model of a navigation system using a graph implementation as underlying data structure. The type of the graph data structure depends on the application scenario of the navigation system. For instance, we need directed, weighted edges for car navigation, but only undirected edges when navigating pedestrians. When implementing feature *Car*, programmers are interested in code fragments of already implemented features to reuse existing functionality. Therefore, the feature context is: $FC = Car$. If we consider the feature context with respect to the given dependencies of the feature model (cf. Figure 1), we can conclude which features must be also considered to identify safely accessible members. We see that each variant, in which feature *Car* is included, also contains feature *Weighted* (cf. Figure 1). This causes the feature *Wgt* to be selected and the feature *Unweighted* to be deselected. Because of these dependencies, further features have to be selected and deselected.

State-of-the-Art Approaches

State-of-the-art approaches are currently conceivable to determine accessible members (i.e., classes, methods, fields) for an SPL and feature context. We analyze the approaches *Feature Module*, *Minimal Variant*, and *Always Available* related to the criteria *completeness* and *soundness*.

- *Completeness:* An approach is *complete* if it contains all members of an SPL that are safely accessible for each feature context.
- *Soundness:* An approach is *sound* if it contains only members that are safely accessible for each feature context.

In general, an incomplete approach hinders code reusing by not showing all safely accessible members of other features. By contrast, an unsound approach can lead to dangling references that result in compile-time errors. Therefore, a developer needs an overview of members that is complete and sound to easily support changes, maintenance, or extensions of SPLs.

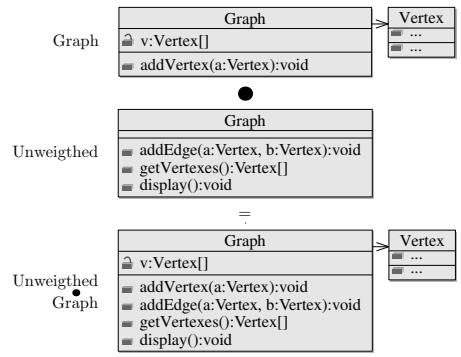

Figure 2: Composition of features *Graph* and *Unweighted*.

Feature Module. One way to identify members that are safely accessible in a given feature context *FC* is to collect all members that are defined in the corresponding feature module *impl*(*FC*) of the feature context. All members that are defined in this feature module are safely accessible and, thus, we classify the approach as sound. However, this approach certainly neglects all members outside of this feature module that are also always safe to access. If there exists a dependency in the feature model that forces the selection of another feature that is outside of *FC* (e.g., a parent feature), the result will be incomplete.

For instance, we are interested in accessible members related to the feature context of *Car* (i.e., $FC = Car$). Using this approach, we look only into the feature module of feature *Car*. This is an easy procedure, but neglects safely accessible members, such as method `addEdge` and `getVertexes` of feature *Weighted*.

Minimal Variant. A state-of-the-art approach to gather safely accessible members is to use the members of certain variants *impl*(v) of an SPL. We generate a variant containing feature *FC* by additionally selecting a minimal number of features to yield a valid variant. For instance, if our feature context contains feature *Graph* (i.e., $FC = Graph$), in addition to other features, we need to select one of the features *Weighted* and *Unweighted* to obtain a valid variant. This decision, however, influences the members present in the resulting variant. If we select feature *Weighted*, method `addEdge(Vertex, Vertex, int)` is included in the resulting set of members, although it is not always safely accessible (i.e., when we decide to select feature *Unweighted*).

The ambiguity to define a variant that can be used as program API is one of the negative aspects of this approach. Furthermore, the manual selection to define a specific variant leads to additional members influencing the soundness. For instance, if we create all variants V_{FC} related to the feature context *FC*, the approach would present only sound results iff the difference of the given variant's members is empty. However, it cannot be ensured that there is no member that is defined only in one of these variants. Therefore, we classify the approach as complete but unsound. Nevertheless, the approach presents complete results, so that

	Incomplete	Complete
Unsound		*Minimal Variant*
Sound	*Feature Module*	
	Always Available	

Table 1: Classification of state-of-the-art techniques.

all safely accessible members are included in the variants of V_{FC}.

To reduce the number of unsound members in the generated variant, it is possible to reduce the number of features that are included in a valid configuration. Therefore, we select only those features that are required to obtain a valid variant. We call this resulting variant – *Minimal Variant*.

Always Available. The approach *always available* uses information given in the feature model to obtain a set of features that are present in all variants such that we can safely rely on the members of these features. An advantage of this approach is the independence to the feature context. We need to determine only the set of core features, which is non-variable and which will only change if the feature model is changed. However, we are unable to determine *all* members that are safely accessible from any given feature context. Therefore, this approach is sound but incomplete.

In Table 1, we summarize the results of our discussion and classify each approach related to the soundness and completeness. In contrast to the presented approaches, we are interested in an approach that is sound *and* complete. Therefore, we introduce feature-context interfaces that fulfill our requirements.

4. FEATURE-CONTEXT INTERFACES

In this section, we present feature-context interfaces; a special view to an SPL's implementation that presents safely accessible members tailored to a given feature context. As a necessary step toward the creation of feature-context interfaces, we explain the *variable interface (VInt)* of an SPL. Afterwards, we provide a definition of feature-context interfaces and describe an algorithm to generate them.

4.1 Variable Interface

The variable interface of an SPL contains all unique signatures of all existing members (i.e., classes, methods, and fields) with a presence condition describing in which features the member is defined. That is, if a member is defined in several modules, it is included in the variable interface only once, but with the list of the defining features (i.e., the presence condition). Because of the presence condition, the existence of each member is variable depending on their definitions in the feature modules. Therefore, we call the resulting structure a variable interface *VInt*.

We specify the signature of each feature module $impl(f_i)$ with $f_i \in F$ as a function *sig*, which takes as input an implementation module $impl(f_i)$ and gives as output the corresponding set of signatures (i.e., classes, methods and fields) in the corresponding implementation module $impl(f_i)$. Accordingly, an SPL's signature M is the union of all signatures of its implementation modules:

$$M = sig(impl(f_1)) \cup ... \cup sig(impl(f_n))$$

```
1: function CreateVariableInterface(F)
2:     VInt := ∅
3:     for f_i ∈ F do
4:         for m ∈ sig(impl(f_i)) do
5:             F_m := getFeatures(VInt, m)
6:             if (F_m ≠ null) then
7:                 F_m := F_m ∪ {f_i}
8:             else
9:                 F_m := {f_i}
10:                VInt := VInt ∪ {(m, F_m)}
11:            end if
12:        end for
13:    end for
14:    return VInt
15: end function
```

Figure 3: Algorithm to create the variable interface.

Now, we can use the set M to define the variable interface denoted as *VInt*. We define the variable interface as a set of tuples (m, F_m), whereas m is a member of M and F_m denotes a subset of all features, in which m is defined. Thus, we define the variable interface as follows:

$$VInt = \{(m, F_m) \mid m \in M, F_m \subseteq F\}$$

Algorithm. In Figure 3, we depict our algorithm to create the variable interface *VInt*. We use the set of features as input from which we can get all signatures of the corresponding feature modules. We calculate the variable interface by starting with an empty set (Line 2). Then, we analyze each member m from the signature of each feature module (Line 3-13). In detail, function `getFeatures` searches for the member's signature (m) in *VInt* and returns the presence condition F_m if it is already defined (Line 5). If the result of function `getFeatures` is defined (i.e., not *null*), we add the feature f_i, in which the current member m was found, to the set of features F_m (Line 7). Otherwise, we create a new set F_m and add feature f_i to F_m (Line 9). Then, we create a new tuple (m, F_m) and add it to the variable interface *VInt* (Line 10). Afterwards, we can continue with the next member.

Example. Let us consider the variable interface of our navigation SPL. Class `Graph` is defined in features *Graph*, *Directed*, *Undirected*, *Weighted*, *Unweighted*, and *Connected*. By contrast, the field `v` of class `Graph` is defined only in feature *Graph*. This leads to the first entries of the following subset of the variable interface (features are represented with underscored characters of Figure 1):

$$\begin{aligned}
VInt = \{ & \\
& (class\ Graph, \{c, d, i, r, u, w\}), \\
& (Graph.v : Vertex[], \{r\}), \\
& (void\ Graph.addVertex(Vertex), \{r\}), \\
& (void\ Graph.addEdge(Vertex, Vertex, int), \{w\}), \\
& (void\ Graph.addEdge(Vertex, Vertex), \{u\}), \\
& (Vertex[]\ Graph.getVertexes(), \{u, w\}), \\
& (void\ Graph.display(), \{c, d, i, u, w\}), \\
& ... \\
& (class\ Vertex, \{c, d, i, r\}), \\
& ... \\
\}
\end{aligned}$$

4.2 Feature-Context Interfaces

A feature-context interface is a non-variable view of the variable interface *VInt*. Variability is not required as the feature-context interface is tailored to a given feature context. This means that the feature context acts as a filter on *VInt* and, thus, the feature-context interface presents only members that are safely accessible in the corresponding feature context. Similar to the variable interface *VInt*, we start with a definition followed by the presentation of an algorithm to compute the feature-context interface.

The feature-context interface *FCI* is a subset of all existing SPL members M (i.e., $FCI_{FM,FC,VInt} \subseteq M$). The presence of each member m depends on the feature model FM, the feature context FC, and the presence condition F_m that has been precomputed in *VInt*.

For the computation of the feature-context interface *FCI*, we filter the existing members given in the variable interface *VInt* using the presence condition of each member relative to a propositional formula of the feature model FM and the feature context FC. That is, all variants that can be generated by selecting the feature of the feature context FC. Therefore, we create a new feature model FM_{FC} that exactly describes those variants.

$$FM_{FC} = FM \wedge FC$$

Afterwards, we check the accessibility of each member m using its presence condition F_m and the propositional formula FM_{FC}. Therefore, we reformulate the presence condition F_m of the current member m into the propositional constraint $Constraint_m$, which is a disjunction of all the features in F_m.

$$Constraint_m = (\bigvee_{f \in F_m} f) \mid (m, F_m) \in VInt$$

For each evaluation of the propositional formula FM_{FC} that leads to a valid variant, we have to check whether at least one of the features in $Constraint_m$ is true. Consequently, we have to check whether the following propositional formula is a tautology:

$$FM_{FC} \rightarrow Constraint_m$$

Based on these definitions, we are able to define the feature-context interface as follows:

$$FCI_{FM,FC,VInt} = \{m \mid (FM_{FC} \models Constraint_m), (m, F_m) \in VInt\}$$

Algorithm. In Figure 4, we show the algorithm that computes a feature-context interface, using the feature model FM, the feature context FC, and the variable interface *VInt* as input. First, we initialize an empty set representing the feature-context interface, in which the result will be stored (Line 2). Afterwards, we create the propositional formula FM_{FC} that represents the input feature model FM with the feature context (Line 5). Based on FM_{FC}, we check which members of the variable interface *VInt* are accessible in the feature context (Line 8-15). Therefore, we create a new propositional formula FM_m for each member's presence condition that can be used to check the member's accessibility. We transform the presence condition F_m of the current member m to the propositional constraint $Constraint_m$ (Line 9) and create an implication of FM_{FC} and $Constraint_m$. If the resulting formula is a tautology, the member is added to the feature-context interface (Line 13).

```
1: function createFCI(FM, FC, VInt)
2:     FCI := ∅ // feature-context interface
3:
4:     // Add the feature context to the feature model
5:     FM_FC := FM ∧ FC
6:
7:     // Check the existence of each member
8:     for (m, F_m) ∈ VInt do
9:         Constraint_m := (⋁_{f∈F_m} f)
10:        FM_m := FM_FC → Constraint_m
11:        // Check for tautology
12:        if (!isSatisfiable(¬FM_m)) then
13:            FCI ← FCI ∪ {m}
14:        end if
15:    end for
16:    return FCI
17: end function
```

Figure 4: Creation of a feature-context interface.

Example. Let us consider our navigation SPL with the feature context $FC = Car$. For each member m in *VInt*, we check the accessibility using the corresponding presence condition of m. For member

$$(Vertex[]\ Graph.getVertexes(), \{u,\ w\}),$$

we create the constraint $Constraint_m$ with $(u \vee w)$. Afterwards, we check whether the formula $FM_{FC} \rightarrow (u \vee w)$ is a tautology. In this case, the formula is a tautology, because in every variable assignment in which FM_{FC} is *true* one of the variable w or u is also *true*. Thus, we add member m to the feature-context interface *FCI*. In the following, we present the feature-context interface of feature *Car* related to our navigation example of Figure 1. We can see that the feature-context interface contains not only members from the features *Weighted* and *Directed*, but also from other features (e.g., field `v` of class `Graph`).

$FCI_{FM,Car,VINT} = \{$
class Graph,
Graph.v : Vertex[],
void Graph.addVertex(Vertex),
void Graph.addEdge(Vertex, Vertex, int),
Vertex[] Graph.getVertexes(),
void Graph.display(),
...further members
class Vertex,
...further classes
$\}$

5. TOOL SUPPORT

We implemented feature-context interfaces as an extension to FeatureIDE. FeatureIDE is an integrated development environment based on Eclipse for SPL development [22]. It is used worldwide by researchers as well as students, and supports a number of composition tools. Our implementation of feature-context interfaces is targeted at the composition tool FeatureHouse [3], which can be used to develop SPLs with feature-oriented programming.

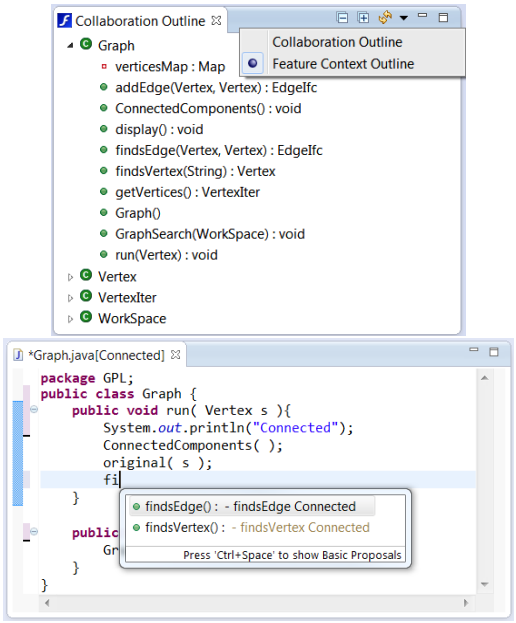

Figure 5: Feature-context outline and content assist for class `Graph` of feature *Connected*.

We use Fuji to identify all members related to all existing feature modules. Fuji is a compiler and type checker for feature-oriented programming [4], which allows us to guarantee the type safe comparison of members that we need to produce a correct variable interface *VInt*. The resulting list of members is stored in FeatureIDE and is the basis to compute a specific feature-context interface. We automatically determine the current feature context by evaluating which class is currently opened in the editor of FeatureIDE and to which feature module this class belongs. Based on the variable interface, the current feature context, and the feature model that is maintained by FeatureIDE, we automatically derive the corresponding feature-context interface. In the following, we present two application scenarios for feature-context interfaces that support an SPL's development. We extended FeatureIDE with an outline view and an auto-completion support.

Feature-Context Outline. Based on feature-context interfaces, we implemented an extended outline view, the *feature-context outline* (see Figure 5). In this outline, we present all classes that are accessible in the feature context of a given feature. Members that are not safely accessible in the current feature context are omitted. We illustrate the functionality using our running example. When we modify class `Graph` of the feature *Connected* the feature context is given by $FC = Connected$, which we use to determine the set of members. As result, we get an overview about all safely accessible members depicted in our feature-context outline.

Content Assist. Beside outline views, Eclipse provides content assists to support developers. FeatureIDE already provides content assist for FeatureHouse based on variants. This content assist helps implementing a feature only for a single variant but can result in compile-time errors in other variants (cf. Section 3). In contrast, we implemented a new content assist based on feature-context interfaces that presents only safely accessible members. In Figure 5, we present the functionality of our content assist. Here, the developer types "`fi`" and the content assist proposes the methods `findsEdge` and `findsVertex`, which can be used in the context of feature *Connected* in class `Graph`.

6. EVALUATION

So far, we argued that feature-context interfaces can be used to prevent errors during the development of SPLs by showing only members to the developer that are safely accessible. Two kinds of evaluations are possible to investigate the potential of feature-context interfaces: a user study and a quantitative evaluation.

We decided to perform a quantitative evaluation, because if we would find no evidence in a quantitative study, a further user study would be pointless. Furthermore, we can draw important insights in which use cases feature-context interfaces reach their full potential. Such conclusions have a broader generality than user studies and represent a basis for future experiments. In our user study, we want to estimate the support for error prevention of feature-context interfaces in relation to state-of-the-art approaches. For this future work, again, we need the results of the quantitative evaluation of this work first.

To quantify the prevention of potential errors of feature-context interface against state-of-the-art approaches (see Section 3), we conducted an empirical study with two *goals* in mind:

(a) We analyze whether state-of-the-art approaches present incomplete or unsound results for existing SPLs.
(b) We quantify how evident potential errors are in existing SPLs.

Hence, we analyze SPLs that were also used in previous research papers.

While state-of-the-art approaches are either unsound (i.e., *minimal variant*) or incomplete (i.e., *always available*, *feature module*) an open question is whether they are also unsound and incomplete in practice. In addition, dependencies between features given in the feature model may influence the available members of a feature context, but it is an open question to which extent.

Subject SPLs. Our evaluation is based on eight feature-oriented SPLs written in Java taken from publicly available repositories.[1] These SPLs are selected from different domains with different sizes in terms of the derivable variants and the number of features. All SPLs exhibit no compile-time errors in all valid variants. Furthermore, these SPLs were used in previous studies by other researchers and can be considered as a benchmark in SPL analysis [3, 15, 23]. In detail, we use two graph libraries `GPL` and `GraphLib`, two games `TankWar` and `GameOfLife`, the text editor `Notepad`, the chat client `PKJab`, the search engine `DesktopSearcher`, and the compression library `ZipMe` as subject SPLs (see Table 2).

6.1 Experimental Design

To compare each state-of-the-art approach to the feature-context interface, we investigate a typical implementation scenario of a developer in which a feature module is to be extended by new functionality.

[1] More information is available on our website (`http://www.fosd.de/multiple/`).

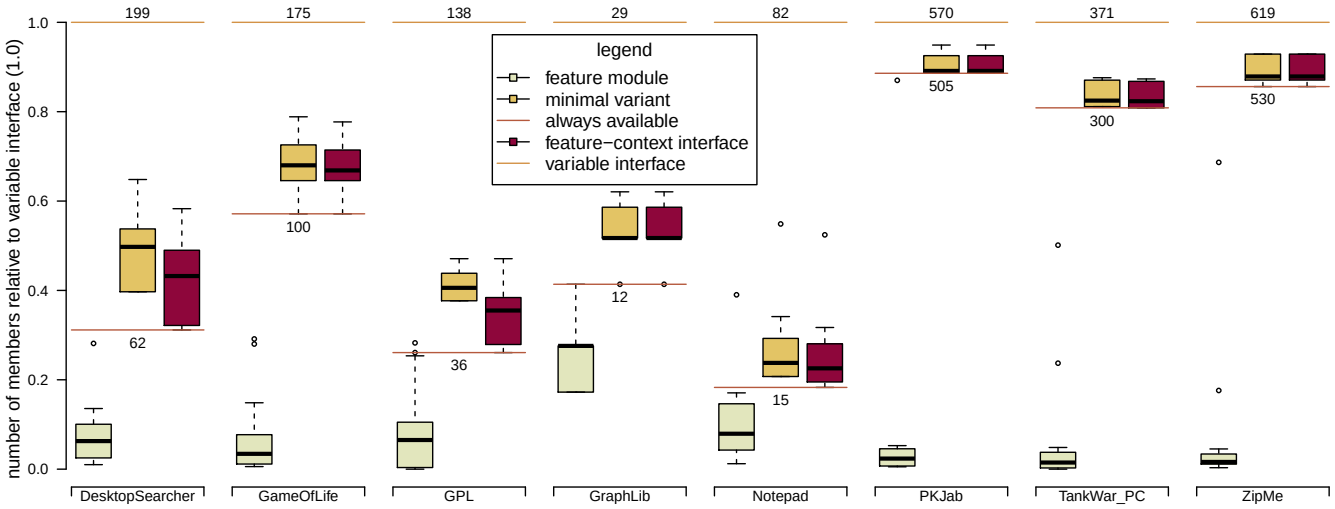

Figure 6: Analysis results of the number of presented members relative (scaled) to the members represented in the variable interface *VInt*.

SPL	Features (Alternative)	Products	Unique Classes
DesktopSearcher	22 (8)	462	21
GameOfLife	23 (2)	65	21
GPL	38 (15)	156	16
GraphLib	6 (0)	16	5
Notepad	15 (4)	512	8
PKJab	12 (0)	48	51
TankWar_PC	37 (7)	87360	21
ZipMe	17 (0)	24	31

Table 2: Investigated product lines of our evaluation (determined by the statistics view of FeatureIDE).

(a) Incomplete or Unsound Results. The experimental procedure to address goal (a) is as follows: we select each feature of an SPL as the feature context and compute for each approach the number of all accessible members. We consider the approaches described in Section 3: *feature module*, *minimal variant*, and *always available*. In addition, we use the feature context to determine the existing members in the feature-context interface. We determine the number of existing members in the variable interface *VInt* and scale the results of all approaches to this set of members, to achieve visually comparable results.

As mentioned in Section 3, the result of the approach *minimal variant* heavily depends on the selection of a specific variant. It is possible that two or more variants are minimal variants related to their number of features but contain different sets of members. To represent the number of members for the approach *minimal variant*, we select for each feature context a minimal variant that results in the minimum number of members.

(b) Potential Errors. The experimental procedure to address goal (b) is similar. First, we compute for each approach and feature context all accessible members. In contrast to procedure (a), we consider not only the number of accessible members, but compare the resulting members directly with the members of the feature-context interface. If a member is available in an approach but not part of the feature-context interface, we count this member as a potential error leading to compile-time errors in the implementation. Thus, all errors that we count for the minimal variant are potential compile-time errors. By contrast, each member that is available in the feature-context interface but not part of a specific approach effectively limits reuse opportunities. Therefore, we count each non-existing member in the approaches *feature module* and *always available* also as a potential error. For illustration purposes, we also scale the number of counted errors for each approach to the number of members in the variable interface *VInt*. This allows us to summarize the counted errors for each approach about all subject systems. As result, we reason about the potential of feature-context interface compared to state-of-the-art approaches.

In contrast to the procedure (a), we investigate all *minimal variants* related to a feature context and include the results in our evaluation. This allows us to identify all possible errors that can occur only in one of all minimal variants.

6.2 Quantitative Results

Based on our experimental design, we also divide the results in two parts that address (a) the completeness and soundness of state-of-the-art approaches and (b) the analysis of potential errors in our subject SPLs.

(a) Incomplete and Unsound Results. In Figure 6, we show the results of our evaluation related to goal (a), in which we analyze the soundness and completeness of each approach. We show the number of all members in the variable interface as the top line in Figure 6. We scale the number of members presented by the other approaches accordingly. The number of members given by the approach *always available* is independent of the feature context and, thus, illustrated as a horizontal line. For example, Notepad has 15 members that are always available, which are 18 % of all members. The results of the other three approaches (i.e., *feature module*, *minimal variant*, and *feature-context inter-*

face) yield one value per feature context. Hence, we show for each approach a box plot indicating the distribution of the number of presented members.[2] That is, a box plot shows for a single approach the distribution of the number of presented members for all feature contexts.

To illustrate the meaning of our results, let us consider the graph product line (GPL). The variable interface *VInt* of the `GPL` contains exactly 138 members (orange line). For the approach *always available*, we identified 36 members, which correspond to 26 % of the members represented in the variable interface *VInt*. Furthermore, the approach *feature module* (lime-green) has a median of 7 %, *minimal variant* (yellow) a median of 41 %, and the approach *feature-context interface* (red) a median of 36 %.

The number of members given in the *feature-context interface* and the *minimal variant* of SPLs `ZipMe`, `PKJab`, and `GraphLib` are equal, and the number of members given in SPL `TankWar_PC` is almost equal. In two systems the approach *feature-context interface* presents noticeably less members as in the corresponding *minimal variants* of a feature context (i.e., `DesktopSearcher` and `GPL`). The SPLs `Notepad` and `GameOfLife` present marginal advantages if a developer uses *feature-context interfaces* instead of *minimal variants*.

We observe that whereas the number of members in the *feature-context interface* and the *minimal variant* often present similar results, the available number of members in the *feature module*s strongly differs. Furthermore, the difference between the median of the *feature-context interface* and the median of the *feature module* can lead to extreme differences, such as SPL `PKJab` which results in 86 % different members.

We further analyze how feature dependencies affect the number of members shown by the different approaches. In Table 2, we present the number of alternative features of each SPL that lead to an interesting finding. The SPLs `ZipMe`, `PKJab`, and `GraphLib` do not contain alternative features. If we correlate this with the results of the approaches *feature-context interface* and *minimal variant* exactly these SPLs exhibit the same number of members. In addition, the SPL `TankWar_PC` contains only two alternative features which is also an SPL in which the results of the *feature-context interface* and the *minimal variant* are almost identical.

(b) Potential Errors. In Figure 7, we present a summary over all SPLs regarding potential errors of the state-of-the-art approaches. Again, we scale the counted errors to the number of presented members in the variable interface. The results are the following: *feature module* (lime-green) has a median of 45 %, *minimal variant* (yellow) has a median of 1 % and *always available* (light-brown) has a median of 6 % of potential errors relative to the number of members given in the variable interface. On average, 6 % of the members in *minimal variants* can cause compile-time errors that we can prevent using the feature-context interface. In addition, Figure 7 shows that the approach *feature module* does not present 52 % members on average. In the worst case, the approach *feature modules* does not present up to 92 % of members that are available in the feature-context interface

[2]We created all box plots with R using the default setting (`http://www.r-project.org/`). The thick black line represents the median of all values. The box itself represents the distribution of 50 % of all values. The whiskers are extended to the data extreme points if they are less than 1.5 times of the interquartile range away from the box. Other points are represented as outliers.

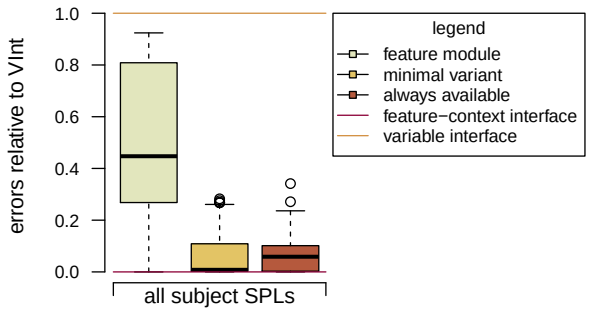

Figure 7: Potential errors of each state-of-the-art approach relative to *VInt* and compared to the feature-context interface.

and, thus, safely accessible. By contrast, the members presented by the approach *always available* causes, on average, 7 % potential errors that can hinder the development of new features. This is significantly less than the potential error caused by the *feature module*, whereas the maximum value of potential errors in *always available* is lower than the mean value of the *feature module*.

6.3 Discussion

Next, we discuss the results of our evaluation according to each technique. Furthermore, we present suggestions for the usage of a specific technique in a particular scenario.

Feature Modules. Also in our subject SPLs, the feature module presents only a small subset of all accessible members in a given feature context. In all evaluated SPLs, this approach does not show even half of all possible accessible members. Although determining the available members is comparatively easy and the approach is classified as sound (cf. Section 3), it is likely that the approach is insufficient in practice.

Minimal Variant. The approach *minimal variant* indicates the best results of all state-of-the-art approaches. In detail, in 3 of 8 SPLs it is sufficient to detect accessible members using the approach *minimal variant*. However, in our SPLs, we can achieve correct results only if there are no alternatives (or constraints acting in a similar fashion) in the feature model. Using alternative features, we would show also members whose usage would end up in compile-time errors, which we also witnessed in our evaluation. Hence, as many feature models contain alternative features [21], we cannot recommend this approach either.

Always Available. Although the approach *always available* presents only parts of all accessible members related to a feature context, this approach could be a good alternative compared to the approach *feature module*. In detail, a developer gets up to 84 % more members compared to the *feature module* (cf. `PKJab`). Hence, the approach *always available* is a sound alternative to using minimal variants. Furthermore, the approach is independent of the feature context and, thus, we can compute always-available members once and not for each feature context.

Feature-Context Interface. The results have shown that there are cases in our subject SPLs in which none of the above approaches is sufficient. Therefore, our approach of

feature-context interfaces is an improvement over state-of-the-art approaches. We have shown that the simple approaches may lead to compile-time errors or that they limit reusability. Compared to the variable interface, it is a non-variable view that requires no further reasoning by programmers. Compared to all other non-variable techniques, it is sound and complete, but also for most of our subject SPLs we experienced superiority.

Based on the evaluation results, we identify two subject systems that can be used for our planed user study. In detail, the SPLs `DesktopSearcher` and `GPL` results in huge differences between the presented members in the *minimal variant* and the *feature-context interface* and, thus, we identified these SPLs as suitable for our user study. By contrast, we have seen that a user study without this evaluation as preliminary step can distort the result using subject systems such as `ZipMe` or `TankWar_PC`. As a further step, we can investigate the impact of feature-context interface on development time.

6.4 Threats to Validity

In the following, we discuss possible threats to validity.

External Validity. External validity refers to how generalizable our results are. We address this threat by first conducting a formal comparison of the feature-context interface with state-of-the-art approaches independent of subject systems (cf. Section 3). Furthermore, we use open-source case studies from a publicly available repository. These case studies have also been used in other research papers [3, 15, 23].

We are aware that the size of the SPLs in terms of number of members can influence the study results. Therefore, we varied the sizes from 29 to 619 members given in the variable interface *VInt*. Having SPLs for various domains, we can draw practical conclusion as to how our *feature-context interface* relate to state-of-the-art approaches.

Internal Validity. Compile-time errors can lead to distorted results related to the computed members of each approach (e.g., by showing false positive members). Therefore, we selected only subject SPLs without compile-time errors.

The correct composition of defined members in each feature module to the variable interface *VInt* strongly depends on the collection and comparison method. So that a string-based method on the source code can lead to an incorrect variable interface. To avoid an incorrect identification of SPL's members, our algorithm is based on the feature-oriented Java compiler Fuji [4]. Using Fuji, we are able to identify all code artifacts that are used in an SPL. In our analysis, we count each defined method and field of a Java-class as a member. During method or field comparisons, we rely again on Fuji. For instance, if we compare the parameters and return types of two methods, Fuji presents us the fully-qualified name of the types. Furthermore, we do not compare methods and fields as Java-method and Java-field signatures. In detail, we use full-qualified names of the member as well as full-qualified names of the parameter and return types.

7. RELATED WORK

Efficient approaches exist for type checking SPLs in feature-oriented programming [2, 15] as well as in annotation-based approaches [12]. These approaches are able to detect errors posterior, whereas we present an approach that is able to prevent some errors a priori during the implementation.

Tools such as *Colored Integrated Development Environment* (CIDE) [11] for annotation-based approaches highlight the source code of individual features using a special color per feature. Furthermore, CIDE present views that involve a special variant, representing a compilable unit (variant view), or represent code of a set of selected features and their needed dependencies (realization view) [14]. The realization view in CIDE is similar to the feature-context interfaces, because they also present code for the currently developed feature. The realization view uses an abstract syntax tree to calculate the visible code, in which each parent of the currently developed code and each mandatory child is presented to the developer. By contrast, the feature-context interface presents not only mandatory children of the abstract syntax tree but presents also optional code by taking the dependencies of the SPL's feature model into account.

Another possibility to filter visibility of code is to use access modifiers, such as private and protected in Java. To overcome the limitation of existing modifiers in feature-oriented programming, Apel et al. present three new modifiers *feature*, *subsequent*, and *program* [4]. For instance, the modifier *feature* restricts the visibility of members to specific features. The developer can restrict the visibility of code manually. Hence, this approach is orthogonal to ours and both approaches can benefit from each other. Therefore, we plan to integrate this functionality in future.

Several approaches exist that address the visibility of features in feature models and present techniques to define views to these models [9, 16, 19]. This way, feature models are tailored to the specific needs of stakeholders. Using these approaches, irrelevant functionality related to the current use case (i.e., context) of the stakeholder can be hidden (i.e., feature-model elements). This is similar to our approach for the implementation of SPLs, in which we hide irrelevant members of code related to the current feature context, but all these approaches do not consider source code.

Kästner et al. present a variability-aware module system that enables to combine variable modules [13]. Their approach includes variability inside modules as well as variability in their interfaces. The module itself can be seen as a product line in which existing variability is partly assigned to the interface of the module, so that configuration options are presented to the interface for other modules. The benefit is that these modules can be type checked without the knowledge of other (variable) modules. The variability in the module interface is comparable with our variable interface, but presents only the external variability that can be used by other modules. While we rely on the variable interface for the calculation of feature-context interfaces, the advantages of feature-context interface is that they show exactly those members being safely accessible. This avoids reasoning on variability for programmers and possibly prevents compile-time errors.

Ribeiro et al. present emergent interfaces, which support safe maintenance of annotation-based SPLs [18]. The authors present *require* and *provide information* to/from other features related to the current part that should be maintained. The selected maintenance part is similar to our described feature context and supports the development of safe annotation-based SPLs. By contrast to our approach that determines accessible members for reuse, the emergent interface presents information based on dataflow analyses that

indicates where the maintained code is reused to prevent breaking other features.

8. CONCLUSION

Software product-line (SPL) engineering facilitates the generation of similar program variants based on a common code base. Although many approaches exist dealing with the implementation of SPLs, their development is still far from trivial, especially for composition-based implementation techniques. Previous research aimed at the detection of compile-time errors, whereas we aim to *prevent* errors. In detail, if the developer knows which code is safe to access related to the currently implemented feature (i.e., the *feature context*), compile-time errors, such as dangling references, can be avoided up-front.

We propose *feature-context interface*s that provide a non-variable view to all safely accessible code in a given feature context. Based on feature-context interfaces, we implemented a content assist and an outline view in FeatureIDE. We evaluated our approach using a set of publicly available SPLs. In these SPLs, we found that only feature-context interfaces can provide exactly all members that are safe to access (not more and not less).

In future work, we extend our tool support for feature-context interfaces so that it can be used as well for annotation-based approaches and for suggestions of new feature dependencies. For instance, if a developer wants to use code that is not accessible in a current feature context, we propose new constraints to the feature model to change the accessibility. Furthermore, based on the results of our evaluation, we plan to conduct a user study to investigate differences in development time and occurrences of errors when using feature-context interfaces compared to state-of-the-art techniques.

Acknowledgments

We thank Sven Apel for suggestions and constructive discussions of previous versions of the paper. In addition, we thank Sebastian Krieter for assisting us with the implementation in FeatureIDE. The work is partially funded by German Research Foundation (DFG), project number SA 465/34-2.

9. REFERENCES

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

[2] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *ASE*, 17(3):251–300, 2010.

[3] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *TSE*, 39(1):63–79, 2013.

[4] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access Control in Feature-Oriented Programming. *SCP*, 77(3):174–187, 2012.

[5] S. Apel, A. von Rhein, T. Thüm, and C. Kästner. Feature-Interaction Detection based on Feature-Based Specifications. *ComNet*, 57(12):2399–2409, 2013.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, 30(6):355–371, 2004.

[7] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[8] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, 2000.

[9] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, and E. Abbasi. Supporting Multiple Perspectives in Feature-Based Configuration. *SoSyM*, 12(3):641–663, 2011.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[11] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.

[12] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *TOSEM*, 21(3):14:1–14:39, 2012.

[13] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *OOPSLA*, pages 773–792. ACM, 2012.

[14] C. Kästner, S. Trujillo, and S. Apel. Visualizing Software Product Line Variabilities in Source Code. In *ViSPLE*, pages 303–313, 2008.

[15] S. Kolesnikov, A. von Rhein, C. Hunsen, and S. Apel. A Comparison of Product-based, Feature-based, and Family-based Type Checking. In *GPCE*, pages 115–124. ACM, 2013.

[16] M. Mannion, J. Savolainen, and T. Asikainen. Viewpoint-Oriented Variability Modeling. In *COMPSAC*, pages 67–72. IEEE, 2009.

[17] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.

[18] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent Feature Modularization. In *SPLASH*, pages 11–18. ACM, 2010.

[19] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-Perspectives on Feature Models. In *MODELS*, pages 252–268. Springer, 2012.

[20] R. Schröter, N. Siegmund, and T. Thüm. Towards Modular Analysis of Multi Product Lines. In *MultiPLE*, pages 96–99. ACM, 2013.

[21] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *ICSE*, pages 254–264. IEEE, 2009.

[22] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *SCP*, 79(0):70–85, 2014.

[23] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *SPLC*, pages 191–200. IEEE, 2011.

[24] F. J. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.