Variability Mining: Consistent Semiautomatic Detection of Product-Line Features

Christian Kästner, Alexander Dreiling, and Klaus Ostermann

Abstract—Software product line engineering is an efficient means to generate a set of tailored software products from a common implementation. However, adopting a product-line approach poses a major challenge and significant risks, since typically legacy code must be migrated toward a product line. Our aim is to lower the adoption barrier by providing semiautomatic tool support—called *variability mining*—to support developers in locating, documenting, and extracting implementations of product-line features from legacy code. Variability mining combines prior work on concern location, reverse engineering, and variability-aware type systems, but is tailored specifically for the use in product lines. Our work pursues three technical goals: (1) we provide a *consistency indicator* based on a variability-aware type system, (2) we mine features at a *fine level of granularity*, and (3) we exploit *domain knowledge* about the relationship between features when available. With a quantitative study, we demonstrate that variability mining can efficiently support developers in locating features.

Keywords—Variability, reverse engineering, mining, feature, software product line, LEADT, feature location.

1 Introduction

Software product line engineering is an efficient means to generate a set of related software products (a.k.a. variants) in a domain from common development artifacts [4]. Success stories of software product lines report an order-of-magnitude improvement regarding costs, time to market, and quality, because development artifacts such as code and designs are systematically reused [4], [49].

Variants in a product line are distinguished in terms of *features*; domain experts analyze the domain and identify common and distinguishing features, such as *transaction*, *recovery*, and different *sort* algorithms in the domain of database systems. Subsequently, developers implement the product line such that they can derive a variant for each feature combination; for example, we can derive a database variant with transactions and energy-saving sort mechanisms, but without recovery. Typically, variant derivation is automated with some generator [17]. Over the recent years, software product line engineering has matured and is widely used in production [4], [49].

Despite this acceptance, adopting a product-line approach is still a major challenge and risk for a company. Typically, legacy applications already exist that must be migrated to the product line. Often companies halt development of new products for months in order to migrate from existing (isolated) implementations toward a software product line [14]. Hence, migration support seems crucial for the broad adoption of product-line technology. Currently, even locating, documenting, and extracting the implementation of a feature that is already part of a single existing implementation is a challenge [25], [28], [42], [43], [60].

Our aim is to lower the adoption barrier of product-line engineering by supporting the migration from legacy code toward a software product line. We propose a system that semiautomatically detects feature implementations in a code base and extracts them. For example, in an existing implementation of an embedded database system, we might want to identify and extract all code related to the transaction feature to make transactions optional (potentially to create a slim and resource-efficient variant, when transactions are not needed). We name this process *variability mining*, because we introduce variability into a product line by locating features and making them variable. Variability mining is one important building block in a larger research context of supporting product-line adoption for legacy applications, others being reengineering of existing variability from *if* and *#ifdef* statements and from program deltas (e.g., branches in a version control system) [21], [23], [35].

A main challenge of variability mining is to locate a feature *consistently* in its *entirety*, such that, after location and extraction, all variants with and all variants without this feature work as expected. In our database example, removing transactions from the system must introduce errors neither in existing variants with transactions nor in new variants without transactions. Unfortunately, full automation of the process seems unrealistic due to the complexity of the task [8]; hence, when locating a feature's implementation, domain experts still need to confirm whether proposed code fragments belong to the feature. We have developed a semiautomatic variability-mining tool that recommends probable code fragments and guides developers in looking in the right location. It additionally automates the tasks of documenting and extracting features.

Mining variability in software product lines is related to research on concept/concern location [8], [19], [22], feature identification [19], [53], reverse engineering and architecture recovery [13], [20], impact analysis [3], [48], and many similar fields. However, there is a significant difference in that variability mining identifies the entire extent of optional (or alternative) features for production use in a product line

• *C. Kästner is with the School of Computer Science at Carnegie Mellon University; A. Dreiling is with the University of Magdeburg and Deutsche Bank AG, Germany; K. Ostermann is with the Department of Mathematics and Computer Science at Philipps University Marburg, Germany.*

instead of locating a concern for (one-time) understanding or maintenance tasks. Detecting features in a product line contributes additional opportunities and challenges, including the following:

1) All variants generated with and without the feature must be correct. We use well-typedness as a *consistency indicator*.
2) Features must be identified at a *fine level of granularity*, because the results of the mining process are used to extract the feature's implementation.
3) Often, developers have *domain knowledge* about existing features and their relationships. If available, this knowledge can be used to improve the mining process.

We implemented and evaluated our variability-mining approach with a tool LEADT for Java code. In a quantitative analysis, we identified 97 % of the code of 19 features in four small product lines. All located features were consistent.

In summary, we contribute: (a) a process to semiautomatically locate, document and extract variable product-line features in a legacy application, (b) a tool for Java code to support the process, (c) a novel use of a variability-aware type system as consistency indicator, (d) an extension of existing concern-location techniques with domain knowledge and fine granularity required in the product-line setting, and (e) a quantitative evaluation with 19 features from 4 product lines.

2 Variability Mining

We define variability mining as the process of identifying features in legacy code and rewriting them as optional (or alternative) features in a product line. We assume that we extract features from a single code base. However, it is not uncommon that, before adopting a product-line approach eventually, developers have already introduced variability in some ad-hoc way that should be migrated as well. For example, developers might have used a *clone-and-own* approach or branches in a version control system, might have introduced variations with command-line parameters or *#ifdef* directives. In such scenario, other complementary migration and reengineering strategies are necessary [1], [21], [23], [35]. Here, we focus only on locating and extracting features from a single legacy-code base (not from deltas between branches). It is difficult to quantify how often locating variability in a single code base is needed in practice. In academia, it is a standard approach to create case studies [16], [25], [28], [42], [43], [58], [60]. From industry, we have anecdotal evidence of similar adoption potential. We do not claim that this adoption strategy is prevalent in practice, but argue that it is a relevant building block in a larger tool box.

Consider the following setting: A company has developed an application and now wants to turn it into a product line. In the product line, several features—that previously existed hidden in the application—should become optional, so that stakeholders can derive tailored variants of the application (with and without these features). In a typical scenario, the company wants to sell variants at different prices, wants to optimize performance and footprint for customers that do not need the full feature set, or wants to implement alternatives for existing functionality.

For illustration purposes, we use a trivial running example of a stack implementation in Java, listed in Figure 1, from which we want to extract the feature *locking* (highlighted), such that we can generate variants with and without *locking*.

```
class Stack {
  int size = 0;
  Object[] elementData = new Object[maxSize];
  boolean transactionsEnabled = true;

  void push(Object o) {
    Lock l = lock();
    elementData[size++] = o;
    unlock(l);
  }
  Object pop() {
    Lock l = lock();
    Object r = elementData[--size];
    unlock(l);
    return r;
  }
  Lock lock() {
    if (!transactionsEnabled) return null;
    return Lock.acquire();
  }
  void unlock(Lock lock) { /*...*/ }
  String getLockVersion() { return "1.0"; }
}
class Lock { /*...*/ }
```

Fig. 1. Example of a stack implementation in Java with feature *locking* (corresponding lines highlighted).

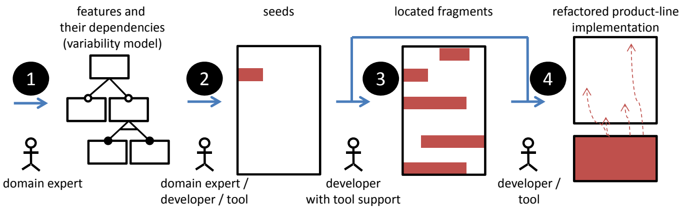
features and their dependencies (variability model)
seeds
located fragments
refactored product-line implementation
1
2
3
4
domain expert
domain expert / developer / tool
developer with tool support
developer / tool

Fig. 2. The feature-mining process.

The variability-mining process consists of four steps as illustrated in Figure 2:

1) A domain expert models the domain and describes the relevant *features and their relationship* in a variability model (the optional and independent feature *locking* in our example).
2) A domain expert, a developer, or some tool identifies initial *seeds* for each feature in the legacy code base. Seeds are code fragments that definitely belong to the feature, such as methods *lock* and *unlock* in our example.
3) For each feature, developers iteratively *expand* the identified code until they consider the feature consistent and complete. Starting from known feature code, the developer searches for further code that belongs to the same feature (all highlighted code in our example).
4) In a final step, developers or tools *rewrite* (or extract) the located code fragments, so variants with and without these code fragments can be generated.

Of course, the process can be executed in an iterative and interleaved fashion. For example, we could start mining a single feature and later continue with additional features, we could add additional seeds later, and we can expand several features in parallel.

Within this process, we focus on the third step of finding all code of a feature. The remaining steps are far from trivial, but are already well supported by existing concepts and tools. In contrast, actually finding the entire implementation of a feature in legacy code currently is a tedious and error-prone task, which we aim to ease with tool support that guides the developer.

We envision a variability-mining tool that recommends code fragments at which the developers should look next. The recommendations are updated whenever additional information is available, such as changes to features and their relationships, seeds, or when developers expand the identified code fragments.

2.1 Existing Support for Variability Mining

While we focus on the third step of finding feature code in this article, we can reuse existing work for the remaining steps of the variability mining process.

Deciding which features to extract (Step 1) is a typical task in product-line engineering that requires communication with many different stakeholders. The decision depends on many influence factors, including many business and process considerations discussed elsewhere [5], [7], [27], [49], [55], [56]. Recently, She et al. even explored extracting variability models from legacy code and other sources [54].

To determine seeds (Step 2), often developers or domain experts can provide hints. Although they might not know the entire implementation, they can typically point out some starting points. Furthermore, search facilities, from simple tools like *grep* to sophisticated information-retrieval mechanisms, such as *LSI* [44], *Portfolio* [45], *SNIFF* [12], and *FLAT*^{3} [53], and analysis tools for configuration parameters [50] can support determining seeds.

Regarding rewrites (Step 4), a simple form of rewriting identified feature code for a product-line setting is to guard located code fragments with conditional-compilation directives, such as the C preprocessor's *#ifdef* and *#endif* directives. Experience has shown that this can usually be done with minimal local rewrites of the source code [29], [58]. More sophisticated approaches refactor the code base and move feature code into a plug-in, an aspect, or a feature module of some form [30], [42], [47]. In prior work, we have shown that such refactoring can be even entirely automated, once features are located in the source code [30].

2.2 Product-Line Context and Design Goals

Finding all code of a feature (Step 3) is related to concern-location techniques and code search engines (e.g., [8], [15], [19], [22], [48], [52], [53]; see Sec. 5 for a more detailed discussion). There is a huge design space for concern-location approaches, many with goals overlapping with ours. In the following, we describe the characteristics of our product-line context and the corresponding design goals for our approach.

Binary and permanent mapping. In a product line, a mapping between features and code fragments is used to drive variant generation. That is, for a given feature selection, a product-line generator automatically derives the corresponding implementation by composing or removing code fragments related to features [17].

The mapping needs to be *binary* in the sense that the mapping between features and code fragments denotes a *belongs-to* relationship, on which we can rely for variant generation. A binary mapping can be used by the generator decide whether to include or exclude a code fragment from a variant; it can also be used as basis for refactorings (Step 4 in Fig. 2). In the product-line context, we speak of *annotations*: a code fragment is annotated with a feature. In contrast, a mere *is-related-to* relationship, possibly with a weight, at method or class level (e.g., "method *push* is likely related to the *locking* concern") is not sufficient for automated variant generation and must be reduced to a binary decision by a developer.

The mapping is *permanent* in the sense that it is not just mere documentation, but actually an integral part of the product line's implementation. Using the mapping between features and code fragments during variant generation is a strong *incentive* for developers to later *update the mapping* when evolving the implementation, preventing erosion often associated with documentation and architecture descriptions.

Due to the goal of binary and permanent mappings, our process is incremental and relies heavily on developer feedback to make the final decisions. However, it also characterizes mining variability as a long-term investment.

Consistency. In a product-line context, whenever we extract a feature, we expect that *all* variants generated with and without that feature must execute correctly. This property gives rise to a *consistency indicator*. When we locate a feature, we need to continue mining, until the feature is located consistently. As a lower bound for a consistency indicator, we require that all variants compile, which we can determine statically. Additionally, we could run a test suite or use some validation or verification methods.

In this paper, we define that a feature is identified *consistently* if all variants are *well-typed*. For example, if we annotated the declaration of *unlock* in Figure 1, but not the corresponding method invocations, then, variants without feature *locking* would be ill-typed and, hence, inconsistent.

Note that consistency does not imply completeness. For example, not annotating class *Lock* would be incomplete but consistent: All variants compile; class *Lock* is just never referenced in variants without *locking*.

Granularity. To achieve consistent and binary mappings, we need to map code fragments at a fine level of granularity. Feature implementations in product lines often consist of small code fragments scattered over multiple classes and methods [29], [40], [48]. This means that we need to be able to annotate even individual statements as we did in Figure 1 (and possibly smaller code fragments).

Domain knowledge. In a product line, domain experts may know some features and their relationship (see Step 1 above). Typical relationships between features are that one feature *requires* another feature (implication) or that two features are *mutually exclusive*. During variability mining, we can exploit such information if available. For example, after identifying feature *locking*, we could identify a mutually exclusive feature *snapshot isolation* (not listed in Fig. 1); during *snapshot isolation*'s identification we can restrict the

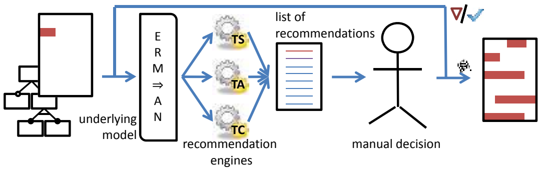

Fig. 3. Recommendation process (part of Step 3)

search space and exclude *locking* code.[1] Similarly, we can exploit *implications* between features (including parent-child relationships) to reduce the search space or to derive additional seeds. For example, before identifying the *locking* feature, we could have already identified a subfeature *dynamicLocking* (ability to disable *locking* at runtime; Lines 4 and 18 in Fig. 1); when subsequently identifying *locking*, we do not need to identify these lines again and can even use them as seeds.

Knowing relationships between features is not necessary for variability mining, but can improve results if available, as we will demonstrate.[2] Describing domain knowledge about variability in variability models and reasoning about it with automated analysis techniques is state of the art in product-line engineering [6], [27].

The four goals—binary and permanent mapping, consistency, fine granularity, and exploiting domain knowledge about feature dependencies—characterize our product-line context. In a sense, variability mining is a process (Steps 1–4) based on concern-location techniques (in Steps 2 and 3), tailored for the need of product-line adoption and the information available in this context.

3 Recommendation Mechanism

To support Step 3 of the variability-mining process, we provide tool support for consistently locating code fragments of a feature. Unfortunately, a full automation of the mining process is unrealistic, so involvement of domain experts is essential. Our semiautomatic variability-mining tool recommends probable code fragments. It guides developers in looking in the right location.

We illustrate the recommendation process in Figure 3. Given domain knowledge (features and their dependencies) and previously located feature code (seeds), our tool builds an internal model of the source code structures, features, and their mappings. Based on that internal model, recommendation engines recommend code fragments that the developer should look at next. Our tool consolidates the recommendations into one prioritized list. Now developers have to decide how to proceed. Developers may reject or accept a recommendation (in a graphical frontend), may change the source code, or may decide to end the location process. After each developer interaction, our tool updates the recommendations to reflect successfully located code fragments, rejected recommendations, changed implementations, and updated domain knowledge.

Since variability mining is a form of concern location tailored for software product lines, we combine existing complementary approaches and enhance them in a product-line–specific way. Each recommendation engine returns a list of recommendations each with a corresponding priority w (range $[0,1]$, reflecting how confident the tool is in the recommendation). Specifically, we develop a mechanism based on a variability-aware type system (to achieve consistency; Sec. 3.2) and combine it with two complementary concern-location mechanisms, topology analysis (Sec. 3.3) and text comparison (Sec. 3.4), known from the literature. The three recommendation mechanisms are complementary; we find more feature code than with each mechanism in isolation, as we will exemplify with our example (Sec. 3.5) and demonstrate empirically (Sec. 4). All mechanisms are based on a variability model and a fairly common, but fine-grained graph of the target program's structure (Sec. 3.1).

3.1 Underlying Model

Before we describe the recommendation mechanisms, we briefly introduce the underlying representation, which represents code elements, features, and relationships between them.

Code elements. To represent code fragments and their relationships, we use a standard graph representation of the source code's structure and dependencies (working on the code's structure instead of textual lines of code). Whereas many concern-location tools (such as Suade [52] and Cerberus [22]) use rather lightweight models and cover only entire methods and fields, we need fine granularity at intraprocedural level, as argued above. For Java, we model compilation units, types, fields, methods, statements, local variables, parameters, and import declarations (we discussed suitable granularity for product lines in prior work [29], [40]; in languages other than Java, we could chose similar structures [32]). Technically, we automatically extract elements from abstract syntax trees provided by Eclipse. We denote the set of all code elements in a program as E.

Between these code elements, we extract relationships ($\mathsf{R} \subseteq \mathsf{E} \times \mathsf{E}$). *Containment* relations describe the hierarchical structure of the code base: a compilation unit contains import declarations and types, a type contains fields and methods, and a method contains statements. *References* cover method invocations, field access, and references to types (as in the return type of a method). Finally, *usage* relationships cover additional relationships when two elements do not directly reference each other, but are used together; examples are assignments, instanceof expressions, and casts.

We extract code elements and relationships from the target code. All relationships mentioned above are based on structural information, type information, and control-flow graph from a compiler frontend that are cheap and precise to compute without data-flow analysis. Also, we explicitly exclude external libraries and assume that the target code is well-typed (although partial models would be possible if necessary [18]).

[1] In a legacy application that was not developed as a product line, mutually exclusive features are less common. They are typically encoded with dynamic decisions, for example, with if-else statements or the strategy design pattern. When migrating the legacy application toward a product line, we can replace the dynamic decisions with compile-time feature selections. The exact process is outside the scope of this paper, but it is important to notice that domain knowledge about mutually exclusive features can be useful conceptually for variability mining nevertheless.

[2] Dependencies usually cover domain dependencies, but may also include known implementation dependencies. Implementation dependencies between features are often discussed as the optional-feature problem and avoided since they reduce variability of the product line [33], [42].

Features. Our product-line setting provides additional domain knowledge that we encode in our model. We describe domain knowledge as a set of features F and relationships between features, extracted from a variability model VM. Although further analysis would be possible, we are interested in two kinds of relationships, mutual exclusion ($M \subseteq F \times F$) and implications ($\Rightarrow \subseteq F \times F$). As explained above, *mutual exclusion* allows us to discard code fragments that already belong to a mutually exclusive feature and *implications* (in the form "feature f is included in all variants in which feature g is included") are useful to provide seeds and because we do not need to reconsider code elements that are already annotated with an implied feature. Implications are especially typical in hierarchical decompositions, in which a child feature always implies the parent feature. We denote the reflexive transitive closure of $\Rightarrow$ by $\Rightarrow^*$. We can either model relationships directly or exact them from other variability model notations (plenty of modeling notations and efficient reasoning techniques have been developed in the product-line community, usually using SAT solvers [6], [46], [57]; in our implementation, we reuse the feature-model editor and the reasoning techniques from FeatureIDE [57]).

Annotations. Finally, we need to model annotations, that is, the mapping between code elements and features. Annotations relate code elements to features ($A \subseteq E \times F$) when assigned by a developer as seed or during the mining process. Additionally, developers can explicitly mark a code fragment as not belonging to the feature, denoted as *negative annotation* ($N \subseteq E \times F$), typically used to discard a recommendation in the mining process.[3] Annotations are always propagated to all children of a code element in the hierarchical code structure. Annotations are used for variant generation in the product line (see *binary mapping* above) and to derive recommendations, whereas negative annotations are used solely as additional input for our recommendation mechanism. Each code element can be annotated with multiple features; in that case, the code element is only included in variants in which all these features are selected (equivalent to nested *#ifdef* directives). In contrast, code used jointly by multiple features is annotated with a separate feature required by the other features through an implication (typically a common parent feature in the variability model).

To consider annotations across multiple features, including available domain knowledge about relationships between features (M and $\Rightarrow$), we introduce the *extent* and the *exclusion* of a feature:

- The *extent* of a feature f is the set of all elements for which we know, from annotations or from domain knowledge, that they belong to f directly or indirectly. An element is in the extent of f, if it is annotated with f or with any feature implied by f.
- The *exclusion* of a feature f is the set of all elements for which we know, from negative annotations or domain knowledge, that they cannot be annotated with f. An element is in the exclusion of f, if (1) there is a negative annotation with f or any feature implied by f or (2) it is annotated with a feature that is mutually exclusive to f.

All code elements that belong neither to the extent of f nor to its exclusion are undecided yet and are candidates for further mining of f.[4]

We define the *extent* and *exclusion* of a feature as

$$extent(f) = \{e \mid (e, f) \in A_{\Rightarrow}\}$$
$$exclusion(f) = \{e \mid (e, f) \in N_{\Rightarrow}\} \cup \bigcup_{(g,f) \in M} extent(g)$$

where $A_{\Rightarrow} = \{(e, f) | (e, g) \in A, g \Rightarrow^* f\}$ and $N_{\Rightarrow} = \{(e, f) | (e, g) \in N, f \Rightarrow^* g\}$ are the closures of A and N with respect to implications.

All recommendation mechanisms use these definitions of *extent* and *exclusion*; hence, they automatically reason about negative annotations and dependencies between features as well. This is the key mechanism to incorporate domain knowledge into recommendations. Like negative annotations, the extent and exclusion of a feature improve recommendations, but are not used for deciding which code fragments to include when generating a variant. When generating variants, we use only annotations (A).

We use the definitions in the remainder of this section to illustrate each recommendation mechanism. In particular, we model each recommendation as a tuple (e, f, w): code element $e \in E$ recommended for feature $f \in F$ with priority $w \in [0, 1]$. Each recommendation mechanisms returns a set of prioritized recommendations: *recommend* $\subseteq E \times F \times [0, 1]$.

3.2 Type System

The type system is our key recommendation mechanism and was the driving factor behind our variability-mining approach. The type system ensures consistency, works at fine granularity, and incorporates domain knowledge about relationships between features.

The underlying idea is to look up references within the product line's implementation as a type system does—references such as from method invocation (source) to method declaration (target), from variable access (source) to variable declaration (target), and from type reference (source) to type declaration (target). We look up references using the relationships R in our model (both at intraprocedural and interprocedural level). If the target of such a reference is annotated with a feature f, but the source of the reference is not part of the extent of f, the type system issues a highly prioritized recommendation to annotate the source—otherwise there are variants without f that include the source but not the target of the reference, hence resulting in a type error (i.e., a violation of our consistency criterion). For example, if method declaration *lock* in our running example

[3] An annotation always maps a feature to an element. If only part of the element belongs to the feature (e.g., a statement in a method or part of an expression in an if statement), only those subelements are annotated. If the used granularity does not expose them as separate elements, the user needs to rewrite the source code as explored elsewhere [31].

[4] In principle, inconsistent annotations are possible. For example, $extent(f)$ and $exclusion(f)$ overlap if a code element is annotated with two mutually exclusive features. Recommendations by the variability-mining tool will not lead to such inconsistencies, but a developer could provoke them manually (by adding incorrect annotations or changing dependencies in the variability model). Our tool could issue a warning in case that happens, so a developer can fix the annotations manually.

(target element) is annotated with feature *locking* whereas the corresponding method invocation in Line 7 (source element) is not annotated, a variant without feature *locking* would result in a method invocation that cannot be resolved.[5]

Already when first experimenting with early versions of the type system over five years ago, we found using type errors for variability mining almost obvious. When annotating a code fragment, say method *lock* in Figure 1, with a feature, the type system immediately reports errors at all locations at which *lock* is invoked without the same feature annotation (Lines 7 and 12 in Fig. 1). We would then look at these errors and decide to annotate the entire statements *Lock l = lock()*, which immediately leads to new type errors regarding local variable *l* (Lines 9 and 14)—note how the type system detects errors even at the fine grained intraprocedural level. This way, with only the type system, we incrementally fix all type errors with additional annotations (or by rewriting code fragments if necessary). With all type errors fixed, we have reached—by our definition—a *consistent* state.

With the type system, we locate feature code (not common code) similar to a low-tech approach: Starting with a compiling implementation that includes the feature, we would comment out all known feature code and compile the remaining implementation. The Java compiler would report type errors regarding unresolvable method invocations and similar errors. We could then comment out (or rewrite) the corresponding locations as well, until the remaining code without the feature compiles (i.e., until we reached consistency). All code we commented out in the process belongs to the feature. Having a type system integrated as a recommender of a variability-mining process improves over the low-tech approach as it can quickly and incrementally recommend additional code fragments, switch between features, and reason about multiple features (and their relationships) at the same time.

We have already implemented such variability-aware type systems for Java and C in prior work (and, for a subset, formally proved that it ensures well-typedness for all variants of the product line) [31], [36]. For variability mining, we reimplemented these checks as recommendation mechanism.

Note how we include domain knowledge about feature relationships by using the extent of a feature, which includes all code elements annotated with implied features: If method declaration and invocation are annotated by different features *X* and *Y*, we do not issue a type error if the invocation's feature *X* implies the declaration's feature *Y*.

We assign the highest priority 1 to all recommendations of the type system, because these recommendations have to be followed in one form or the other to reach a consistent state. Still, in isolation, the type system is not enough for variability mining. It ensures consistency, but is usually insufficient to reach completeness; more on this in Section 3.5.

Conceptually, we can formalize our type checker as a function that takes the type relationships R in a program and returns recommendations with priority 1 for all code elements e and features f that are referenced by other annotated elements:

$$\mathit{recommend}_{TS} = \big\{(\mathsf{e},\mathsf{f},1) \mid (\mathsf{e},\mathsf{e}') \in \mathsf{R} \wedge \mathsf{e} \notin \mathit{extent}(\mathsf{f}) \;\wedge\; \mathsf{e}' \in \mathit{extent}(\mathsf{f})\big\}$$

3.3 Topology Analysis

Next, we adopt Robillard's topology analysis [52] and adjust it for the product-line setting (fine granularity, domain knowledge). The underlying idea of topology analysis is to follow the graph representation of the system from the current extent to all structural neighbors, such as called methods, structural parents, or related variables in an assignment. Then, the algorithm derives priorities and ranks the results using the metrics *specificity* and *reinforcement*. The intuition behind *specificity* is that elements that refer to (or are referred from) only a single element are ranked higher than elements that refer to (or are referred from) many elements. The intuition behind *reinforcement* is that elements that refer to (or are referred from) many annotated elements are ranked higher; they are probably part of a cluster of feature code.

The algorithm follows all relationships R in our model. For example, it recommends a method such as *lock* in Figure 1, when the method is mostly invoked by annotated statements (reference relationship in R); it recommends a local-variable declaration such as *l* in Figure 1, when the variable is only assigned from annotated code elements (usage relationship in R); and it recommends an entire class, when the class contains mostly annotated children (containment relationship in R).

We calculate the priority with weight_{TA}, closely following Robillard's algorithms. We adapt it only for the product-line setting: First, we determine relationships at all levels of granularity supported by our model (i.e., down to the level of statements and local variables), whereas Robillard considered methods and fields only. Second, we consider relationships between features (domain knowledge, if available) by using the entire *extent* of a feature (which includes annotations of implied features, cf. Sec. 3.1) instead of only directly annotated code fragments. In addition, we reduce the priority of a recommendation if an element refers to (or is referred from) elements that are known as *not* belonging to the target feature (negative annotations) or that belong to mutually excluded features: We simply calculate the priority regarding all excluded elements $\mathit{exclusion}(\mathsf{f})$ and subtract the result from the priority regarding the extent:

$$\begin{aligned}&\mathit{recommend}_{TA} = \big\{(\mathsf{e},\mathsf{f},w) \mid \mathsf{e} \in \mathit{neighbors}(\mathit{extent}(\mathsf{f})),\\ &w = \mathit{weight}_{TA}(\mathsf{e},\mathit{extent}(\mathsf{f})) - \mathit{weight}_{TA}(\mathsf{e},\mathit{exclusion}(\mathsf{f}))\big\}\end{aligned}$$

The definition of weight_{TA} can be found in the appendix.

3.4 Text Comparison

Finally, we use text comparison to derive recommendations between declarations of methods, fields, local variables, and types. Text comparison is not restricted to neighboring elements as our type system and the topology analysis are. The general idea is to

[5] In fact, type checking is more complicated when language features such as inheritance, method overriding, method overloading, and parameters are involved. Also feature dependencies beyond implications can be considered. For such cases, we adjusted the type system's lookup functions and check implications between the variability model and annotations using a SAT solver. To understand the recommendation mechanism, the simple model described here is sufficient; for details, we refer the interested reader to our formal discussions in [31].

tokenize declaration names [11] and to calculate the importance of each substring regarding the feature's *vocabulary*. The vocabulary of a feature consists of all tokens in $\mathit{extent}(f)$. Intuitively, if many annotated declarations contain the substring "lock" (and this substring does not occur often in $\mathit{exclusion}(f)$), we recommend also other code fragments that contain this substring.

We use an ad-hoc algorithm to calculate a relative weight for every substring in our vocabulary. We count the relative occurrences of each substring (i.e., occurrences of a token divided by the overall number of tokens) in declarations in $\mathit{extent}(f)$ and subtract the relative occurrences in $\mathit{exclusion}(f)$. That is, negative annotations and annotations of mutually exclusive features (see definition of *exclusion* above) give negative weights to words that belong to unrelated features. By using $\mathit{extent}(f)$ and $\mathit{exclusion}(f)$, we again consider domain knowledge for calculating recommendation priorities (if available).

We implemented our own mechanisms, because it was sufficient to experiment with an additional text-based recommendation mechanism. Our simple implementation was already able to improve recommendations. Nevertheless, for future versions, we intend to investigate tokenization, text comparison, and information retrieval more systematically and potentially use ontologies and additional user input to characterize a feature's vocabulary more accurately.

$$\begin{aligned}&\mathit{recommend}_{TC} = \\ &\quad \{(e, f, \mathit{weight}_{TC}(e, \mathit{vocb}(\mathit{extent}(f)), \mathit{vocb}(\mathit{exclusion}(f))))\}\end{aligned}$$

Additional explanations for weight_{TC} and *vocb* can be found in the appendix.

3.5 Putting the Pieces Together

For each code element, we derive an overall recommendation priority w_* by merging the priorities w_{TS}, w_{TA}, and w_{TC} of the three complementary recommendation mechanisms for this code fragment (we assume a priority of 0 if a recommendation engine does not recommend this code fragment). Following Robillard [52], we use the operator $x \uplus y = x + y - x \cdot y$ to merge priorities in a way that gives higher priority to code fragments recommended by multiple mechanisms; the operator yields a result that is greater than or equal to the maximum of its arguments (in the range $[0, 1]$). For an element recommended by all three mechanisms, we calculate the overall priority based on the three priorities of the respective recommendations: $w_* = w_{TS} \uplus w_{TA} \uplus w_{TC}$.

To illustrate the complementary nature of the three comparison mechanisms, consider our initial stack example in Figure 1 once more.

- The type system recommends many code fragments that are critical by definition, because they must be annotated (or rewritten) to achieve consistency. In our example, the type system recommends the invocations of method *lock* in Lines 7 and 12 with priority 1, once the corresponding method declaration (Lines 17–20) is annotated; the invocation would also be identified by the topology-analysis mechanism and text comparison, but with lower confidence.
- In contrast, the type system would not be able to identify the field declaration of *transactionsEnabled* in Line 4, because removing the reference without removing the declaration would not be a type error; it just results in dead code.[6] In this case, also text comparison would fail without additional ontologies, because it would not detect the semantic similarity between the tokens *transaction* and *locking*. Nevertheless, topology analysis contributes a recommendation, because the field is only referred to from annotated code fragments leading to a high reinforcement score.
- Finally, neither type system nor topology analysis would recommend the method declaration *getLockVersion* that is part of the interface but never called from within the implementation; here, our text comparison provides a suitable recommendation.

This example illustrates the synergies of combining the three complementary recommendation mechanisms, where each mechanism can recommend additional code fragments that another mechanism might not find. In addition, our tool is extensible; so, we could easily integrate additional recommendation mechanisms, for example recommendations based on dynamic execution traces or data-flow properties.

4 Evaluation

Our goal is to evaluate to which degree recommendations from our tool guide developers to consider code fragments that are actually part of a feature's implementation.[7]

4.1 Implementation

We have implemented our variability-mining solution—system-dependency model, type system, topology analysis, and text comparison—as an Eclipse plug-in called LEADT (short for *Location, Expansion, And Documentation Tool*) for Java, on top of our product-line environment CIDE [29]. LEADT reuses CIDE's infrastructure for variability modeling and reasoning about dependencies, for the mapping between features and code fragments, and for extraction facilities, once code is annotated (see Sec. 2.1). Code elements and their relationships are extracted from Eclipse's standard JDT infrastructure. LEADT and CIDE are available online at http://fosd.net/ and can be combined with other tools on the Eclipse platform.

Product-line developers using LEADT follow the four steps outlined in Section 2 (Fig. 2):

1) Modeling features and their relationships (as far as known) in CIDE's variability-model editor.
2) Manually annotating selected seeds, possibly with the help of other tools in the Eclipse ecosystem.
3) Expanding feature code, possibly following LEADT's recommendations. LEADT provides a list of prioritized

[6] Technically, the field is unused code but not dead in the sense that a compiler would report, because the field is visible outside the class and is part of the class' interface.

[7] Initially, we considered also a comparison with other concern-location tools (see Sec. 5). However, since the tools were designed for different settings and use different levels of granularity, such comparison would always be biased by the setting and the different goals of the individual tools.

recommendations for each feature. Developers are free to explore and annotate any code (or undo annotations in case of mistakes), but will typically investigate the recommendations with the highest priority and either annotate a corresponding code fragment or discard the recommendation by adding a negative annotation (or even annotate the code fragment with a different feature). After each added annotation, LEADT immediately updates the list of recommendations. Since LEADT can only judge consistency but not completeness, developers continue until they determine that a feature is complete. We will discuss reasonable stop criteria below.

4) Rewriting the annotated code (optional), possibly using CIDE's facilities for automated exports into conditional compilation and feature modules [30].

Again, the process supports iteration and interleaving. Developers can stop at any point, add a feature or dependency, undo a change, rewrite the source code, or continue expanding a different feature. In each case, LEADT updates its internal model on the fly and provides recommendations for the current context.

4.2 Case Studies

Before quantitatively evaluating the quality of LEADT's recommendations, we briefly summarize experience from two qualitative case studies. First, using a think-aloud protocol, we observed a developer while he mined variability in the database management system HyperSQL (160 000 lines of Java code). Second, replicating a previous decomposition, we performed variability mining for four features from the diagramming application ArgoUML (305 000 lines of Java code). To us, the case studies serve two purposes: (a) they provide insights in how developers interact with LEADT in practice and (b) they informally explore benefits and limitations of variability mining. Even though we attempt to take a neutral approach, case studies provide insights into individual cases and are not suited or meant as objective generalizable evaluation.

In the HyperSQL case study, we observed how a developer interacts with the tool during variability mining, mining three features with 248 to 2819 lines of code over four hours. The developer, an experience PhD student in databases, was familiar with the domain, but not with the particular code base. We could observe that the developer easily understood the recommendation mechanism and appreciated tool support for such repetitive tasks. He quickly trusted the recommendations. He mostly followed the recommendations but looked also at the broader context of recommended code, sometimes using also the search function within a file. He still always came back to the recommendations eventually. In the process he refactored 5 local code fragments. As most interesting insights in usability we learned that we should provide functionality to postpone the decision on a recommendation that is currently not obvious and to sort recommendations not only by priority but also by locality, so that a user can investigate all recommendations within a file before jumping to the next file.

In the ArgoUML case study, we had a different focus. We selected ArgoUML because it was previously decomposed manually by others [16] and because it allowed us to explore variability mining at large scale. Without looking at the previous decomposition beyond feature names, we extracted four features with up to 1245 annotations and up to 37 649 lines of code. In the process, we made errors, reverted changes, and continued without problems; during the mining process, we switched back and forth between features as convenient and refactored 4 code fragments. In an ex-post analysis, comparing to the previous manual decomposition, we discovered that we identified exactly the same set of elements for one feature and a superset for the other three features. It is not always easy to draw the line when code should belong to a feature, and different domain experts could defend different opinions. All additional code that we identified was left as dead code in the previous decomposition (for which topology analysis and text comparison provide recommendations, see Sec. 3.5). That is, depending on the interpretation, we found more feature code or made a different judgment; we argue that our tool-supported decomposition yielded a better result. More details on both case studies, including a closer analysis of the differences between our and the previous decomposition, can be found in an accompanying technical report [34].

To check consistency, we compiled all resulting variants. In addition, we also executed the applications and their existing test suites on selected variants. All variants were well-typed (i.e., consistent) and all tested variants passed all test cases, except for test cases that specifically addressed the removed feature (for example, 141 out of 1192 test cases in ArgoUML referred to activity diagrams and could not be executed when the feature was not selected).

Although the case studies provide some interesting insights about how developers use our tool, it is difficult to measure the quality, impact, or completeness of recommendations objectively. Different developers may have different opinions about the scope of a feature, which again might easily be influenced by a wide range of observer-expectancy effects. Such human influence can easily lead to biased results and reduce internal validity. Therefore, we restricted our report of the case studies here to the essential experience and focus on a controlled quantitative evaluation that we describe next.

4.3 Quantitative Evaluation

To evaluate the quality of LEADT's recommendations quantitatively, we measure *recall* and *precision* in a controlled setting. *Recall* is the percentage of found feature code, compared to the overall amount of feature code in the original implementation. *Precision* is the percentage of correct recommendations compared to the number of inspected recommendations.

4.3.1 Study Setup and Measurement

The critical part of an experiment measuring recall and precision is to find a suitable oracle that defines ground truth for the correct mapping between code fragments and features. An incorrect oracle would lead to incorrect results for both recall and precision.

To combat experimenter's bias, we do not design the oracle ourselves or rely on domain experts that might be influenced by the experimental setting. Instead, to find oracles, we followed two strategies: (a) we searched for programs that were previously decomposed by other researchers and (b) we use existing product lines, in which the original product-line developers already established a mapping between code fragments and features using *#ifdef* directives (independently of our analysis). In existing product lines, we use the code base without any annotations as starting point to re-discover all features.

Our strategies exclude experimenter bias, but limit us in our selection of oracles. Already in concern-location research, realistic and reliable oracles are rare [19]; in the product-line context existing oracles are even harder to find. We cannot simply use any large scale Java application, as we did when locating new features in HyperSQL (see case studies), because they do not have oracles available. Similarly, we cannot use any previous case studies that were created with an early version of the type system in CIDE, such as BerkeleyDB [29]. The resulting trade-off between internal validity (excluding bias) and external validity (many and large studies) is common for decisions in experimental design. Even though that meant resorting to comparably small systems with only few domain dependencies, after our case studies, we decided to emphasize internal validity in our quantitative evaluation.

After selecting the oracles, the evaluation proceeds as follows, following the process illustrated in Figure 2. As first step, we create a variability model in LEADT, reusing the names and dependencies from the oracle. As second step, we add seeds for each feature (see below). The third step performs the actual iterative expansion process, one feature at a time: We take the recommendation with the highest priority (in case of equal priority, we take the recommendation that suggests the larger code fragment). If, according to the oracle, the recommended code fragment belongs to the feature, we add an annotation; otherwise, we add a negative annotation.[8] We iteratively repeat this process until there are no further recommendations or until we reach some stop criteria (see below). Instead of actually extracting code in a fourth step, we determine recall by comparing the resulting annotations with the oracle and precision by comparing the numbers of correct and incorrect recommendations. Finally, we continue the process with the next feature.

With this process, we exclude all human influence. With the oracle, we emulate a developer that always makes perfect decisions for the given recommendations, thus eliminating the problem of human judgment errors and misinterpretations of recommendations. While this is not entirely realistic (external validity), it makes measurement objective, repeatable, and automatable.

There are different strategies to determine seeds. To exclude experimenter bias, we use a conservative strategy based on an existing tool—the information retrieval engine of FLAT*³* [53] (essentially a sophisticated text search; cf. Sec. 5). To determine a single seed per feature, we start a query with the feature's *name* (assuming the name reflects the domain abstraction). FLAT*³* returns a list of methods and fields, of which we use the first result that is correct according to our oracle (a field, a method, or all feature code inside a found method). We discuss the influence of different or more seeds in Section 4.3.4.

Deciding when to stop the mining process for a feature (stop criterion) is difficult, as the developer cannot compare against an oracle. Possible indicators for stopping are (a) low priority of the remaining recommendations and (b) many incorrect recommendations in a row. In our evaluation, we stop the mining process after ten consecutive incorrect recommendations. We discuss alternative stop criteria in Section 4.3.4.

To understand the subtleties of our metrics for recall and precision, it is important to keep the evaluation process in mind. The tool always recommends a single code element at a time, but potentially at different granularity, such as an entire class or a single statement. In our process, deciding whether a recommendation is correct is a binary decision, there is no partial credit for recommending a class of which only two statements belong to a feature. Since developer decisions are emulated to be objective, the process will always annotate a subset of the oracle's feature code, never too much. We measure recall in lines of code, based on the original layout of the source code, which generally follows the Java conventions in all projects. In contrast, we measure precision by counting recommendations considered in our iterative process: Which percent of recommendations was accepted as correct before the stop criterion has been reached. The exact definitions are:

$$\text{Recall} = \frac{\textit{Lines of code annotated when stop criterion reached}}{\textit{Lines of code annotated in the oracle}}$$

$$\text{Precision} = \frac{\textit{Correct recommendations}}{\textit{All recommendations investigated before stop criterion}}$$

4.3.2 Oracles

We selected four different oracles developed by others, covering academic and industrial systems, and covering systems developed with and without product lines in mind.

- **Prevayler.** The open-source object-persistence library Prevayer (8009 lines of Java code, 83 files) was not originally developed as a product line, but has been manually decomposed into features at least three times [25], [42], [58]. Prevayler makes a perfect oracle for variability mining, because all previous decompositions agree almost perfectly on the extent of each feature and because Prevayler was *not* developed as a product line. We use a version that was annotated, independent of our variability-mining research, by de Oliveira at the University of Minas Gerais, Brazil (PUC Minas) with five features: *Censor*, *Gzip*, *Monitor*, *Replication*, and *Snapshot*, with the dependency

[8] Actually, mirroring human behavior (experienced in HyperSQL and others), when a specific recommendation is correct, we also look at the directly surrounding code elements and annotate the largest possible connected fragment of feature code. For example, if the tool correctly recommends a statement and the oracle indicates that the entire method belongs to the feature, we assume that a developer would notice this and annotate the entire method. Technically, we recursively consider siblings and parents of the recommended code element up to compilation-unit level. In addition to the described "greedy" approach, we measured also a conservative one in which we annotate *only* the recommended element. Compared to the conservative approach, our greedy approach improves overall recall from 84 to 97 %, decreases precision from 65 to 42 %, and requires 3.7 times less iterations. We argue that the greedy approach is more realistic; hence, we do not further discuss results from the conservative approach.

		Feature Size			Mining Results		
Project	Feature	LOC	FR	FI	IT	Recall	Prec.
Prevayler	Censor	105 (1 %)	10	5	32	100 %	41 %
	Gzip	165 (2 %)	4	4	27	100 %	18 %
	Monitor	240 (3 %)	19	8	53	100 %	42 %
	Replication	1487 (19 %)	37	28	64	100 %	67 %
	Snapshot	263 (3 %)	29	5	47	81 %	46 %
MobileM.	Copy Media	79 (2 %)	18	6	33	97 %	26 %
	Sorting	85 (2 %)	20	6	36	96 %	46 %
	Favourites	63 (1 %)	18	6	31	100 %	43 %
	SMS Transfer	714 (15 %)	26	14	44	100 %	62 %
	Music	709 (15 %)	38	16	51	99 %	59 %
	Photo	493 (11 %)	35	13	55	99 %	49 %
	Media Transfer	153 (3 %)	4	3	25	99 %	13 %
Lampiro	Compression	5155 (12 %)	33	20	42	100 %	66 %
	TLS Encryption	86 (0 %)	13	6	24	81 %	29 %
Sudoku	Variable Size	44 (2 %)	5	4	24	100 %	29 %
	Generator	172 (9 %)	9	7	29	98 %	42 %
	Solver	445 (23 %)	40	12	46	100 %	58 %
	Undo	39 (2 %)	5	4	29	100 %	21 %
	States	171 (9 %)	26	7	43	99 %	52 %

LOC: lines of code (and percentage of feature code in project's code base);
FR: Number of distinct code fragments; FI: Number of files; IT: Number of iterations

TABLE 1
Feature characteristics and mining results.

Censor → *Snapshot*. In the search for additional oracles, we investigated several manual decompositions of other projects (including AgroUML discussed above), but none of them had similar quality; none were verified or repeated independently.

- **MobileMedia.** Developed from scratch as medium-size product line at the University of Lancester, UK with 4653 lines of Java ME code (54 files) [24], MobileMedia contains six features, *Photo*, *Music*, *SMS Transfer*, *Copy Media*, *Favourites*, and *Sorting*, with the following dependencies: *Photo* ∨ *Music* and *SMS Transfer* → *Photo*.[9] We added a feature *Media Transfer* and the dependency *Media Transfer* ↔ (*SMS Transfer* ∨ *Copy Media*) to the variability model, which are used later to annotate code common to the two transfer features (which is implemented in the original implementation with *#ifdef SMS* || *Copy*). Unfortunately, FLAT[3] would not find any feature code for *Media Transfer*, but thanks to domain knowledge about feature dependencies, we could mine it without seeds (cf. Sec. 4.3.4). Despite being a medium-sized academic case study, MobileMedia makes a suitable oracle, because its Java ME code is well maintained and peer reviewed [24] and used in many other studies. The analyzed version was implemented with conditional compilation; so, we derived a base version by removing all preprocessor directives.
- **Lampiro.** The open-source instant-messaging client Lampiro, developed by Bluendo s.r.l. with 44 584 lines of Java ME code (147 files),[10] provides variability using conditional compilation, like MobileMedia. Of ten features, we selected only two: *Compression* and *TLS Encryption* (without dependencies), because the remaining features were mere debugging features or affected only few code fragments in a single file each (finding a seed would be almost equivalent to finding the entire extent of the feature). Lampiro is interesting as oracle for an industrial product line, in which features were implemented by the original developers with conditional compilation.
- **Sudoku.** The small *Sudoku* implementation (1975 lines of Java code, 26 files), result of a student project at the University of Passau in Germany, contains five features: *States* (for saving and restoring games), *Undo*, *Solver*, *Generator*, and *Variable Size*. The project was designed and implemented as a product line, but using a composition-based approach [2], from which we reconstructed a common base version and corresponding annotations. Despite the small size, we selected *Sudoku* for a particular characteristic of its implementation: the features have dependencies and incrementally extend each other. Specifically, there are the following dependencies: *Generator* → *Solver*, *Solver* → *Undo*, and *Undo* → *States*.

In Table 1, we list some statistics regarding lines of code, code fragments and affected files for each of the 19 features, to give an impression of their complexity and their scattered nature. Overlapping between features (corresponding to nested *#ifdef*) is quite common, but unproblematic; we simply need to locate such code fragments for each feature. All oracles are available (e.g., for replication or comparison) in LEADT's repository.

4.3.3 Variability-Mining Results

In Table 1, we list the number of iterations (i.e., number of considered recommendations) and the measured recall and precision for each feature. On average, we could locate 97 % of all code per feature, with an average precision of 42 %. The results are stable, independent of the kind of oracle (academic vs. industrial, legacy application vs. existing product line).

The high recall shows that we can find most features almost entirely, even with our conservative single seed per feature. Although not all features have been located entirely, all identified features are still consistent; we successfully compiled all variants (40 in MobileMedia, 24 in Prevayler, 4 in Lampiro, and 10 in Sudoku). When manually investigating all missing feature code, we found that it is usually not connected to the remaining feature code. Specifically we found the following common scenarios:

- *Connected only by string literal.* In *MobileMedia*, menu options and event processing are connected only by string literals: An event loop dispatches events depending on the caption of the menu option. For example, although our algorithm recommended the event processing code for "Copy" in feature *Media Transfer*, it did not find the corresponding button declaration with the same string literal. In principle our text-comparison recommender

[9] Source code: http://mobilemedia.cvs.sf.net, version 6_OO, last revision Oct. 2009. We use the feature names published in [24], which abstract from the technical feature names used for implementation, just as a domain expert would. For example, it uses "Music" instead of the implementation flag "includeMMAPI". Furthermore, we added a missing dependency SMSTransfer → Photo to the variability model, which we detected in prior work [31].

[10] http://lampiro.bluendo.com/; Lampiro version 9.6.0 (June 19th, 2009) available at http://lampiro.googlecode.com/svn/!svn/bc/30/trunk/.

could handle such cases, but it currently compares only identifiers, no literals.

- *Extended interface.* In several cases a feature adds additional functionality to the public interface of the program, which might be called by other programs using the code as library. The corresponding methods are never called from within the feature or from any other code in the program. For example, in feature *Snapshot* in *Prevayler*, the feature stores an extra value in a field and provides three additional public methods to the database's API. Although the field is initialized by feature code, the priority was too low to be detected before our stop criterion. Similarly, feature *Generator* in *Sudoku* adds a public method *setInitial* which makes internal program state shared by all variants mutable to external users.
- *Independent change.* Feature *Photo* changes the label of a menu option from "Exit" to "Back" by reassigning the caption in an extra constructor statement. The changed caption then triggers different behavior in the event loop (a different interaction mode with dialogs). There is no visible trace in the source code that could map the constructor statement to the feature; not even the assigned string "Back" gives a hint.

Investigating the missing feature code, we found that it is usually not connected to the remaining feature code (dead code of the feature or isolated methods) or connected only by string literals (text comparison currently only compares definitions not literals).

At first sight, the precision of our approach appears to be quite low. However, considering our goal to guide developers to probable candidates, the results illustrate that following recommendations by LEADT is by far better than searching at random (which would yield a precision equal to the relative amount of feature code shown in the LOC column; differences are strongly significant according to a *t*-test for paired samples). In addition, keep in mind that our stop criterion demands at least ten incorrect recommendations, because developers would not necessarily know that they found the entire feature after few correct steps. For example, the 29 % precision of feature *Variable Size* results from four correct recommendations, which find the entire feature code, followed by 10 incorrect recommendations to reach the stop criterion. Not considering the last ten incorrect recommendations would improve the overall average precision from 42 to 76 %.

4.3.4 Further Measures

Beyond our default setting, we investigated the influence of several parameters more closely. Due to space limitations, we provide only a brief overview and omit details.

Influence of domain knowledge. Although not directly visible from Table 1, domain knowledge about *dependencies* between features can have a significant impact on the results of the mining process.[11] The influence of knowing dependencies becomes apparent when mining features in isolation or in a different order. For features without dependencies, the order does not influence the results at all, but for features with dependencies it does. We selected the order in Table 1 such that, in case of a dependency $A \rightarrow B$, feature A is mined before B. As explained in Sections 2 and 3.1, known dependencies can increase the extent (or exclusion) of a feature and improve the mining results. The influence is visible for all features with dependencies. Mining those features in isolation leads to lower precision for *Snapshot* (34 %), *Photo* (33 %), *States* (30 %), *Undo* (12 %), and *Solver* (35 %) and leads to lower recall for *States* (68 %). In addition, for features implied from other features, we can yield similar results without providing seeds at all. This was especially convenient for feature *Media Transfer*, for which we could not determine seeds with FLAT*3*, but which we could still mine because of known dependencies.

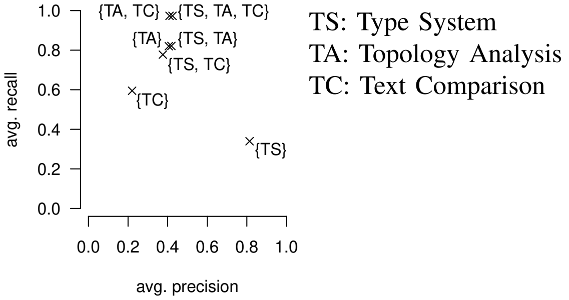

Fig. 4. Combining recommendation mechanisms.

Importance of the recommendation mechanisms. The recommendation mechanisms contribute to the results to different degrees. By rerunning the evaluation in different configurations, we explored different combinations of the recommendation mechanisms and plot the resulting average recall and precision over all case studies in Figure 4. Especially type system and text comparison are not effective on their own. As predicted in Section 3.5, the mechanisms are complementary—combining them improves performance.

Although we cannot provide a fair direct comparison to other concern-location tools (since the tools were developed for different purposes and evaluated with different measures, the setup would always introduce bias toward one solution), the comparison of recommendation mechanisms gives an insight into conceptual differences between tools that use different recommendation mechanisms (see Sec. 5).

For instance just using the type system achieves only a comparably low recall; it benefits significantly from combination with other recommenders. A simple low-tech approach that relies purely on compiler errors while removing feature code (see Sec. 3.2) would achieve a similarly poor performance as the type system in isolation. The dead code found in a prior manual decomposition of our ArgoUML case study (see Sec. 4.2) can be interpreted as confirmation.

We might interpret the results as showing that topology analysis and text comparison are sufficient and that the type system contributes very little beyond them. This would however discard a noticeable quality difference hidden in the aggregated numbers: Recommendations of the type system are made with

[11] The influence of domain knowledge about mutually exclusive features conceptually also has an influence. However, we could not evaluate that influence in our setup, because none of the oracles had mutually exclusive features. Also, mining mutually exclusive features typically requires to rewrite code fragments during the mining process, which does not fit with our automated evaluation in which we excluded all human experimenter bias.

higher confidence and are always actionable. The type system in isolation nearly reaches a precision of 100 %, limited only by incorrectly recommending code of dependent features (that is, for feature f with f ⇒ c, the system recommends a not-yet-annotated code fragment that actually belongs to a feature c; a developer would probably recognize the problem and annotate the code with the correct feature). In the context of our mechanically evaluated quantitative study, the type system's higher precision has only a minimal effect when combined with topology analysis, because the type system issues a smaller number of recommendations (101 recommendations, 19 % of all recommendations) and because the topology analysis makes similar recommendations eventually. Still, we conjecture that the different recommendation confidence influences developers when reasoning about a given recommendation, because the type system's high-confidence recommendations typically represent obvious cases violating the consistency criterion that are easy to decide.

More or other seeds. In principle, the selection of seeds can have a strong influence on the performance of the variability-mining process. However, we found that already with a single seed, we can achieve very good results. In addition, we found that the results are quite stable when selecting other seeds. Using the second, third, fourth, or fifth search result from FLAT^3, instead of the first, hardly changes the result. Only few seeds from FLAT^3 (about one out of ten) lead to a significantly worse result, otherwise recall is mostly the same and also precision deviates only slightly. Using the five first results combined as seed, yields similar or slightly better results than those in Table 1. Also handpicking larger seeds, as a domain expert might do, leads to a similar recall, usually found in less steps. This shows that the recommendation mechanisms are quite efficient finding connected fragments of feature code, almost independent of where the mechanisms start.

Stop criterion. Finally, we have a closer look at the stop criterion. Note that we selected our stop criterion before our evaluation; although we could determine a perfect criterion ex-post, we could not generalize such criterion. In Figure 5, we plot the average recall and precision for mining all 19 features with different stop criteria. We can observe that up to five incorrect recommendations in a row are quite common and should not stop the mining process, whereas continuing after more than eight incorrect recommendations hardly improves recall further (at the cost of lowered precision). In addition, we checked an alternative stop criterion based on the priority of the next recommendation. We can observe that only looking at recommendations with the highest priority 1.0 already is sufficient for 70 % recall, but even recommendations with priority 0.3 contribute to the mining process. Of course a combination of both criteria is possible, but we conclude that already the simple "10 consecutive incorrect recommendations" seems to be a suitable (slightly conservative) stop criterion.

4.4 Threats to Validity

Our case studies explore feature mining in a realistic setting, but may be biased regarding the experimenter's and subject's decisions and knowledge about the system. Hence, we do not attempt to generalize, but interpret the results as encouraging experience report only.

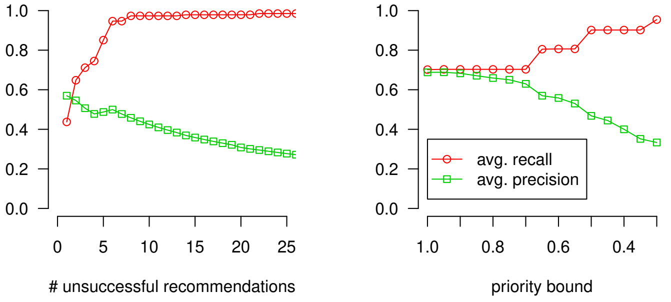

Fig. 5. Alternative Stop Criteria.

In our quantitative evaluation, we attempted to maximize internal validity and exclude bias as far as possible by selecting neutral oracles. Regarding external validity, the selection of four rather small oracles with few features each still does not allow generalizing to other software systems. Furthermore, the selection of some existing product lines as oracles could introduce new bias: Potentially, because the system was already implemented as a product line, it might use certain implementation patterns for product lines. We are not aware of any confounding pattern in the analyzed systems though and results of all case studies, including Prevayler, align well. None of our oracles contained mutually exclusive features. We determined small seeds conservatively and mechanically using FLAT^3, which may not reflect a developer's strategy, and we set an artificial stop criteria, but evaluated the impact of each. Also, in our evaluation setting, we emulated developers to always make perfect decisions, greedily annotating the largest possible code fragments but not too much (see above), whereas in practice developers will sometimes make mistakes and revert annotations.

Regarding internal and construct validity, the common measures recall and precision both depend on the quality of the oracles. We have carefully selected oracles as discussed above (e.g., Prevayler was decomposed several times independently), but the oracles may still contain errors. Our definition of recall measures lines of code of Java code (default formating) instead of counting structural elements. Lines of code are more intuitive to interpret than measures of the codes internal hierarchical structure. Because our tool works on code elements (see Sec. 3.1) but our recall metric measures lines of code, technically, there is the possibility for inaccuracy in which multiple code fragments share a single line, of which only some are annotated. This corner case never occurred in our evaluation.

5 Related Work

Variability mining is related to asset mining, architecture recovery, concern location, and their related fields; it tries to establish stable traceability links to high-level features for extraction and variant generation and combines several existing approaches. However, variability mining is tailored to the specific challenges and opportunities of product lines (consistency indicator, fine granularity, domain knowledge).

The process of migrating features from legacy applications to a product line is sometimes named *asset mining* [5], [7], [55], [56]. Whereas we focus on technical issues regarding locating, documenting, and extracting source code of a feature, previous work on asset mining focused mostly on process and business considerations: when to mine, which features to mine, or whom to involve. Therefore, they weight costs, risks, and business strategy, and conduct interviews with domain experts. Their process and business considerations complement our technical contribution.

Architecture recovery has received significant attention [20]. Architecture recovery extracts traceability links for redocumentation, understanding, maintenance, and reuse-related tasks; usually with a long-term perspective. It typically creates traces for coarse-grained components and can handle different languages. Fine-grained location of features is not in the scope of these approaches.

Work on *aspect mining* searches crosscutting concerns in the source code and aims to extract them into separate aspects [38]. In the location phase, aspect mining often focuses more on finding repeating structures in the source code (homogeneous crosscutting, clone detection), though it also employs concern-location techniques (see below). Once concerns are identified, the extraction is similar to our approach, and in fact, we could use existing aspect-oriented refactorings [47] as subsequent rewrite techniques as mentioned in Sec. 2.1.

Code search engines like *Google Code Search* and more sophisticated tools, such as *LSI* [44], *Portfolio* [45], *SNIFF* [12], and *FLAT*^{3} [53] use various techniques to find code fragments related to a user query. Developers may search for code fragments in different context, for example, when trying to understand a code fragment, when searching for a reusable code fragment for a specific problem, or when identifying code of a specific concern. Sophisticated code search engines use information retrieval techniques often enhanced by reasoning about the underlying structure of a code fragment and about its context, similarly to some of our recommendation engines; McMillan et al. provide a good overview of different technical approaches [45]. Overall, code search engines typically return (a list of) individual code fragments ('snippets'), which often are a starting point for further investigation. In contrast, the goal of variability mining is to find the entire extent of a feature, typically consisting of many code fragments, as described in Sec. 2.2, wherein code search engines may be a good starting point to determine seeds. In our evaluation, we used *FLAT*^{3} as neutral seed generator.

There is a vast amount of research on (semi-)automatic techniques to locate concerns, features, or bugs in source code, known as concept assignment [8], concern location [22], feature location [53], impact analysis [48], or similar. Throughout the paper, we have used the term *concern location* to refer to all of these related approaches. A typical goal is to understand an (often scattered) subset of the implementation for a maintenance task. Examples are locating the code responsible for a bug or determining the impact of a planned change. Similar to architecture recovery, concern location approaches establish traceability links between the implementation and some concepts that the developer uses for a specific task.

Many different techniques for concern location exist: there are static [8], [48], [52] as well as dynamic [15], [59] and hybrid [22], [53] techniques, and techniques that employ textual similarity [22], [53] as well as techniques that analyze static dependencies or call graphs [8], [22], [45], [48], [52] and program traces [15], [22], [53], [55]. Many approaches work at method granularity [22], [45], [52], [53], but also fine-grained approaches have been investigated [48], [59]. For a comprehensive overview, see recent surveys [15], [19]. Many approaches complement ours and can be extended for a product-line setting. Due to space restrictions, we focus on four static concern-location approaches that are closely related to our approach: *Suade*, *JRipples*, *Cerberus*, and *Gilligan*.

We adopted Robillard's *topology analysis* in *Suade* [52] for variability mining. Topology analysis uses static references between methods and fields to determine which other code elements might belong to the same concern. *Suade* uses heuristics, such as "methods often called from a concern's code probably also belong to that concern," and derives a ranking of potential candidates. As explained in Section 3.3, we extended Suade's mechanism with domain knowledge and use a more fine-grained model to include also statements and local variables.

Petrenko and Rajlich's ripple analysis in *JRipples* similarly uses a dependency graph to determine all elements related to given seeds [48]. A user investigates neighboring edges of the graph manually and incrementally (investigated suggestions can lead to new suggestions). *JRipples* lets the user switch between different granularities from class level down to statement level. In that sense, *JRipple*'s granularity matches that of variability mining, but *JRipple* has no notion of consistency or domain knowledge.

Cerberus combines different techniques including execution traces and information retrieval, but introduces an additional concept called *prune-dependency analysis* to find the complete extent of a concern [22]. Prune-dependency analysis assigns all methods and fields that reference code of a concern to that concern. For example, if a method invokes transaction code, this method is assigned to the transaction concern as well. The process is repeated until concern code is no longer referenced from non-concern code. *Gilligan* combines a similar prune-dependency analysis with Suade's topology analysis [26]. Gilligan is tailored specifically for reuse decisions, to locate and copy code excerpts from legacy code. This scenario is similar but requires different decisions. A key decision in Gilligan is when *not* to follow a dependency and replace a method call with a stub in the extracted code. For instance, if the code of a located concern calls a library function, the Gilligan user decides whether the library should be extracted together with the concern's code or whether to extract incomplete code, which will be completed in the context where the extracted code will be reused. In contrast, in a product line, we would decide whether the library should be always included as part of the common base program or whether it should belong to the feature code (in which case no other nonfeature code is allowed to call the library). When considering only a single concern at a time, prune-dependency analysis in Cerberus and Gilligan is similar to a simple form of our variability-aware type system (and similar to a low-tech approach depending on compiler errors, see Sec. 3.2). However, our type system is more fine-grained and additionally considers domain knowledge about relationships between features.

Beyond traditional concern-location techniques, *CIDE+* is the closest to our variability-mining concept [58]. In parallel to our work, the authors pursued the same goals of finding feature code at fine granularity in a single code base. They even built upon the same tool infrastructure (CIDE [29]). In contrast to our approach, they solely use a type-system-like mechanism, along the lines of Cerberus' prune-dependency analysis [22], but do not connect their work with additional concern-location techniques and do not exploit knowledge about feature dependencies. Instead, they focus more on automation and propose few but large change sets, whereas we provide many atomic recommendations to developers. They have evaluated their approach measuring precision and recall on Prevayler and AgroUML, but the reported numbers are not comparable, because CIDE+ uses a different interaction model and large seeds ($> 80\,\%$ of the feature code for most features in Prevayler, instead of a single field or method as seed in our work). In Section 4.3.4, we have shown how our integration of concern-location techniques (a) yields better results than using only a type system and (b) renders the process less fragile to the selection of seeds.

Instead of a tool comparison, we evaluated different conceptual strategies in Section 4.3.4 (Fig. 4). A low-tech manual approach iteratively using only compiler errors would probably be less convenient (see Sec. 3.2) and achieve a result conceptually similar {TS}, but would require many invocations of the compiler in different steps and different variants. *CIDE+* is conceptually equivalent to that approach but essentially automates the manual interaction with the compiler. *Suade* is conceptually similar to {TA} and *Gilligan* to {TS, TA}.

Finally, there have been many efforts of migrating legacy applications to product lines. Where we start with a single legacy application and identify features that have already been implemented but not made optional, others have focused on reengineering existing variability from program deltas (e.g., branches in version control systems) [21], [23] and reengineering existing variability from *#ifdef* directives [1], [35]. We argue that variability mining is one important complementary building block in a larger tool set of adoption and migration strategies for product-line technology in the presence of legacy applications.

6 Conclusion

Software product lines are of strategic value for many companies. Because of a high adoption barrier with significant costs and risks, variability mining supports a migration scenario in which features are extracted from a legacy code base, by providing semiautomatic tool support to locate, document, and extract features. Although we use existing concern-location techniques, we tailor them to the needs of software product lines (consistency indicator, fine granularity, domain knowledge). We have demonstrated that variability mining can effectively direct a developer's attention to a feature's implementation.

In future work, we intend to explore synergies with further recommendation mechanisms. Especially, it would be interesting to investigate recommendation engines based on dataflow and information-flow properties that have been recently explored for product lines [9], [10], [41]. Our evaluation setup also enables to study robustness of our approach against incorrect user inputs, especially incorrect decisions on a recommendation. Another interesting direction is to use consistency criteria beyond well-typedness, such as running test suites or verifying specifications for all variants, which could exploit recent approaches to symbolically execute tests in all variants of a product line [37], [39], [51].

Acknowledgments.

We are grateful to Norbert Siegmund for sharing his experience with HyperSQL, to Eyke Hüllermeier for hints regarding measures in our experiment, and to Paolo Giarrusso, Sven Apel, and the anonymous reviewers for helpful comments on prior drafts of this paper. Käster and Ostermann's work was supported in part by ERC grant #203099. Dreiling's work was supported by the Metop Research Institute.

References

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can We Refactor Conditional Compilation into Aspects? In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 243–254. 2009.

[2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231. 2009.

[3] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[5] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel. Transitioning Legacy Assets to a Product Line Architecture. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 446–463. 1999.

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

[7] J. Bergey, L. O'Brian, and D. Smith. Mining Existing Assets for Software Product Lines. Technical Report CMU/SEI-2000-TN-008, SEI, 2000.

[8] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The Concept Assignment Problem in Program Understanding. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–498. 1993.

[9] E. Bodden. Position Paper: Static Flow-Sensitive & Context-Sensitive Information-flow Analysis for Software Product Lines. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2012.

[10] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 13–24. 2012.

[11] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the Tokenisation of Identifier Names. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 130–154. 2011.

[12] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering*, pages 385–400. 2009.

[13] E. J. Chikofsky and J. H. C. II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7:13–17, 1990.

[14] P. Clements and C. W. Krueger. Point/Counterpoint: Being Proactive Pays Off/ Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, 2002.

[15] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Trans. Softw. Eng. (TSE)*, 35(5):684–702, 2009.

[16] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 191–200. 2011.

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

[18] B. Dagenais and L. Hendren. Enabling Static Analysis for Partial Java Programs. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 313–328. 2008.

[19] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53Ű95, 2012.

[20] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Trans. Softw. Eng. (TSE)*, 35:573–591, 2009.

[21] S. Duszynski, J. Knodel, and M. Becker. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 303–307. 2011.

[22] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 53–62. 2008.

[23] D. Faust and C. Verhoef. Software Product Line Migration and Deployment. *Software: Practice and Experience*, 33(10):933–955, 2003.

[24] E. Figueiredo et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 261–270. 2008.

[25] I. Godil and H.-A. Jacobsen. Horizontal Decomposition of Prevayler. In *Proc. IBM Centre for Advanced Studies Conference*, pages 83–100. 2005.

[26] R. Holmes, T. Ratchford, M. Robillard, and R. Walker. Automatically Recommending Triage Decisions for Pragmatic Reuse Tasks. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 397–408. 2009.

[27] K. 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, SEI, 1990.

[28] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 223–232. 2007.

[29] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. 2008.

[30] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 157–166. 2009.

[31] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(3):Article 14, 2012.

[32] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, pages 175–194. 2009.

[33] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 181–190. 2009.

[34] C. Kästner, A. Dreiling, and K. Ostermann. Variability Mining with LEADT. Technical Report 01/2011, Department of Mathematics and Computer Science, Philipps University Marburg, 2011.

[35] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. 2011.

[36] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 2012.

[37] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD)*, pages 1–8, 2012.

[38] A. Kellens, K. Mens, and P. Tonella. A Survey of Automated Code-Level Aspect Mining Techniques. *Transactions on Aspect-Oriented Software Development*, 5490(IV):143–162, 2007.

[39] C. H. P. Kim, S. Khurshid, and D. Batory. Shared Execution for Efficiently Testing Product Lines. In *Proc. Int'l Symp. Software Reliability Engineering (ISSRE)*, pages 221–230. 2012.

[40] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. 2010.

[41] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, 2013. to appear.

[42] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–121. 2006.

[43] R. Lopez-Herrejon, L. M. Mendizabal, and A. Egyed. Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 181–190. 2011.

[44] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 214–223. 2004.

[45] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding Relevant Functions and Their Usage. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 111–120. 2011.

[46] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 231–240. 2009.

[47] M. P. Monteiro and J. M. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 111–122. 2005.

[48] M. Petrenko and V. Rajlich. Variable Granularity for Improving Precision of Impact Analysis. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 10–19. 2009.

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

[50] A. Rabkin and R. Katz. Static Extraction of Program Configuration Options. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 131–140. 2011.

[51] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 445–454. 2010.

[52] M. P. Robillard. Topology Analysis of Software Dependencies. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 17(4):1–36, 2008.

[53] T. Savage, M. Revelle, and D. Poshyvanyk. FLAT^3: Feature Location and Textual Tracing Tool. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 255–258. 2010.

[54] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse Engineering Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 461–470. 2011.

[55] D. Simon and T. Eisenbarth. Evolutionary Introduction of Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 272–282. 2002.

[56] C. Stoermer and L. O'Brien. MAP – Mining Architectures for Product Line Evaluations. In *Proc. Working Conf. Software Architecture (WICSA)*, page 35. 2001.

[57] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 254–264. 2009.

[58] M. T. Valente, V. Borges, and L. Passos. A Semi-Automatic Approach for Extracting Software Product Lines. *IEEE Trans. Softw. Eng. (TSE)*, 38(4):737–754, 2012.

[59] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[60] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 188–205. 2004.

APPENDIX

We calculate $\mathit{weight}_{TA}(e, X)$ where X is a set of relevant elements (either the extent or the exclusion of f) as follows:

$$\mathit{targets}(e) = \{e' | (e, e') \in R\}$$

$$\mathit{sources}(e) = \{e' | (e', e) \in R\}$$

$$\mathit{weight}_{TA}(e, X) = \frac{1 + |\mathit{targets}(e) \cap X|}{|\mathit{targets}(e)|} \cdot \frac{|\mathit{sources}(e) \cap X|}{|\mathit{sources}(e)|}$$

Functions *targets* and *sources* determine the neighboring elements in the program structure (relation R). The weight is high if a large percentage of sources and targets are within the set of relevant elements X (i.e., already annotated elements or elements known to be in conflict). In each fraction, the denominator describes specificity and tends to lower the priority if there are many neighboring elements, whereas the numerator describes reinforcement and increases the priority if many neighboring elements are already in the set of relevant elements. The two fractions account for incoming and outgoing relationships in R. See [52] for a detailed explanation.

Function *vocb* of the text-comparison mechanism can be explained conceptually as follows: It receives a set of code elements and tokenizes their names and removes stop words. It counts how often each token occurs, relative to the total number of tokens. Subsequently, $weight_{TC}$ compares the tokens of a code element with the weighted tokens in both vocabularies:

$$weight_{TC}(e, \nu_1, \nu_2) = \sum_{t \in tokenize(e)} (\nu_1(t) - \nu_2(t)) \cdot \rho(t)$$

$\nu(t)$ denotes the lookup of the relative frequency of a token in a vocabulary and ρ is a weight that we use to give lower priorities for shorter tokens (0 for length 1, .33 for length 2, .67 for length 3, and 1 for all longer tokens in our evaluation). The function cuts off values below 0 and above 1. Since both vocabularies typically have many tokens with comparably low weights priorities given by the text-comparison mechanisms tend to be rather low, which reflects the low confidence we have in this mechanism.

Christian Kästner Christian Kästner is an assistant professor in the School of Computer Science at Carnegie Mellon University. He received his PhD in 2010 from the University of Magdeburg, Germany, for his work on virtual separation of concerns. For his dissertation he received the prestigious GI Dissertation Award. His research interests include correctness and understanding of systems with variability, including work on implementation mechanisms, tools, variability-aware analysis, type systems, feature interactions, empirical evaluations, and refactoring.

Alexander Dreiling Alexander Dreiling received a Master's degree in Business Information Systems from Otto-von-Guericke University Magdeburg, Germany in 2010. His research focused on variability and software product lines, feature-oriented software development, and feature location. Alexander Dreiling currently works as Assistant Vice President at Deutsche Bank AG, Group Technology, in post merger integration projects as project manager.

Klaus Ostermann Klaus Ostermann is a professor of computer science at the Philipps University of Marburg, Germany. His main research interests are in programming languages and software engineering for modular software development.