Variability Mining with LEADT

Christian Kästner
Philipps University Marburg, Germany

Alexander Dreiling
University of Magdeburg, Germany

Klaus Ostermann
Philipps University Marburg, Germany

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 extends prior work in three important aspects: (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.

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 [2]. 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 [2, 30].

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. Over the recent years, software product line engineering has matured and is widely used in production [2, 30].

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 [9]. 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 [16, 19, 24, 25, 39].

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 and extracting features.

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 [6]; 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 [6, 14], feature identification [32], reverse engineering and architecture recovery [8, 13], impact analysis [29], and many similar fields. However, there is a significant difference in that variability mining identifies optional (or alternative) features for production use in a product line instead of locating a concern for (one-time) understanding or maintenance tasks. Detecting features in a product line contributes three opportunities and challenges often not required in previous work:

1. All variants generated with and without the feature must be executable. This provides us a *consistency indicator*.
2. Features must be identified at a *fine level of granularity* (e.g., identify statements inside a method), 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. We conducted a series of case studies to evaluate practicality of variability mining. In a quantitative analysis, we identified 97 % of the code of 19 features in four product lines. All located features were consistent.

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

```
1  class Stack {
2    int size = 0;
3    Object[] elementData = new Object[maxSize];
4    boolean transactionsEnabled = true;
5
6    void push(Object o) {
7      Lock l = lock();
8      elementData[size++] = o;
9      unlock(l);
10   }
11   Object pop() {
12     Lock l = lock();
13     Object r = elementData[--size];
14     unlock(l);
15     return r;
16   }
17   Lock lock() {
18     if (!transactionsEnabled) return null;
19     return Lock.acquire();
20   }
21   void unlock(Lock lock) { /*...*/ }
22   String getLockVersion() { return "1.0"; }
23 }
24 class Lock { /*...*/ }
```

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

2 Variability Mining

We define variability mining as the process of identifying features in legacy code[1] and rewriting them as optional (or alternative) features in a product line. 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 Fig. 1, from which we want to extract the feature *locking* (highlighted), such that we can generate variants with and without *locking*.

The variability-mining process consists of four steps:

1. A domain expert models the domain and describes the relevant *features and their relationship* (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 (i.e., 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

[1] We assume that we extract features from a single code base; when development already is branched to implement ad-hoc variability with a *clone-and-own* approach (not uncommon before adopting a product-line approach eventually), other complementary mining strategies based on software merging are necessary. Still, locating features in each individual clone can be a useful preparation.

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, instead of providing all features and their relationships upfront, we could start mining a single feature and later continue with additional features.

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 tasks, 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 annotate additional code fragments.

2.1 Existing Support for Variability Mining

For the remaining steps of the variability mining process, we can combine existing results.

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 [3, 5, 18, 30, 34, 35]. Recently, She et al. even explored extracting variability models from legacy code and other sources [33].

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* [26] and *FLAT*^{3} [32], 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 [20, 38]. More sophisticated approaches refactor the code base and move feature code into a plug-in, an aspect, or a feature module of some form [21, 24, 28]. In prior work, we have shown that such refactoring can be even entirely automated once features are located in the source code [21].

2.2 Product-Line Specifics

Finding all code of a feature (Step 3) is related to concern-location techniques (e.g., [6, 10, 14, 29, 31, 32]; see Sec. 5 for a more detailed discussion). However, we found that the product-line setting of variability mining differs significantly from the concern location.

Permanent mapping. Whereas results of concern location are usually shown for a one-time understanding or maintenance task, variability mining extracts features in a product line driving variant generation. That is, the extracted information are not mere

documentation but actually become part of the product line's implementation. Inaccurate feature location may hamper maintenance, but for generating variants with and without a feature, a *precise* and *complete* mapping between features and code fragments is crucial.

Actually using the mapping between features and code fragments during variant generation is a strong *incentive* for developers to *update the mapping* when evolving the implementation. Hence, erosion of the mapping over time is not as problematic as in concern mappings or architecture descriptions. Mining variability can be considered as long-term investment.

Consistency. Whenever we extract a feature, we expect that all variants generated with and without that feature must execute correctly, which 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 Fig. 1, but not the corresponding method invocations, then, due to dangling method invocations, 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 consistency, we need an expressive mechanism to map features to code fragments precisely and at a fine level of granularity, because feature implementations often consist of small code fragments scattered over multiple classes and methods [20, 23, 29]. For instance, it is not sufficient to annotate only entire classes or methods. The information that "method *push* is related to the *locking* feature" is not precise enough. Instead, annotations need to describe a definite *belongs-to* relationship, on which we can rely for variant generation. This also means that we need to be able to annotate individual statements as we did in Fig. 1 (or even smaller code fragments).

Domain knowledge. In a product line, domain experts often know features and their relationship (or this information can be recovered in preliminary interviews with stakeholders [3,5,34,35]). 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 search space and exclude *locking* code.[2] 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;

[2] 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 for variability mining nevertheless.

Lines 4 and 18 in Fig. 1); when subsequently identifying *locking*, we do not need to identify these lines again and can actually use them as seeds.

Knowing relationships between features is not necessary for variability mining, but can improve results if available, as we will demonstrate. Describing them in variability models and reasoning about them with automated analysis techniques is state of the art in product line engineering [4, 18].

The four differences—permanent mapping, consistency indicator, fine granularity, and domain knowledge—separate variability mining from traditional concern-location techniques. In principle, variability mining could also be used for classic concern location, but the additional overhead for requiring consistency, for a permanent mapping, or for variability modeling might be too high for some maintenance tasks. In a sense, variability mining is a concern-location process specialized for the need of product-line adoption, which exploits the additional 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 still necessary. However, given domain knowledge (features and their dependencies) and previously located feature code (seeds), our semiautomatic variability-mining tool recommends probable code fragments. It guides developers in looking in the right location. During the mining process, our tool constantly updates the recommendation to reflect already located code fragments and updated domain knowledge.

Since variability mining is a form of concern location specialized for software product lines, we combine existing complementary approaches and support them with product-line–specific enhancements. We focus on the mechanism that derives recommendations and their priorities (range $[0, 1]$). To this end, 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 mechanism, topology analysis (Sec. 3.3) and text comparison (Sec. 3.4), known from the literature. Combining the three mechanisms exploits synergies; we find more feature code than with each mechanism in isolation (Sec. 3.5). All mechanisms are based on a variability model and a fairly common, but fine-grained system-dependency graph of the target program (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 model of a static system-dependency graph between source-code elements. Whereas most concern-location tools (such as Suade [31] and Cerberus [14]) use rather lightweight models and cover only entire methods and fields, we need finer granularity at intraprocedural level, as argued above. For Java, we model compilation units, types, fields, methods, statements, local variables, parameters, and import declarations.[3] We denote

[3] We decided not do include elements at finer granularity, such as parts of expressions, because they are difficult to handle in many product-line implementations; it is usually easier to rewrite the source code locally,

the set of all code elements in a program as E.

Between these code elements, we extract relationships ($R \subseteq E \times E$). *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. We explicitly exclude external libraries, and we assume that the target code is well-typed (although partial models would be possible if necessary [12]).

Features. In contrast to traditional concern-location techniques, 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. We assume a consistent variability model with at least one valid feature combination. Although further analysis would be possible, we are interested in two kinds of relationships, mutual exclusion ($M \subseteq F \times F$) and implications ($\Rightarrow \subseteq F \times F$). *Mutual exclusion* allows us to discard code fragments that already belong to a mutually exclusive feature. *Implications* (in the form "feature f is included in all variants in which feature g is included") are helpful because we do not need to reconsider code elements that are already annotated with an implied feature and, additionally, we can use them as seeds. 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 $\Rightarrow^*$. An automatic extraction of relationships between features from a variability model is straightforward and efficient with reasoning techniques developed in the product-line community, usually using SAT solvers [4,27,37]. In our implementation, we reuse the feature-model editor and the reasoning techniques from FeatureIDE [37].

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. Annotations are used for variant generation in the product line (or for rewriting the located code into a product-line implementation) 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).

The *extent* of a feature f is the set of elements that is already known to belong to f, whereas its *exclusion* is the set of elements that can never belong to the feature due to negative annotations and mutual exclusion between features.

In the product-line setting, annotations are especially interesting when considering domain knowledge about features: We define the closure of A and N with respect to implications as $A_{\Rightarrow} = \{(e,f) | (e,g) \in A, g \Rightarrow^* f\}$ and $N_{\Rightarrow} = \{(e,f) | (e,g) \in N, f \Rightarrow^* g\}$, respectively. Using the definitions from above, we define the extent

for example, by splitting an expression into multiple statements. We avoid a more detailed discussion on suitable granularity and defer the interested reader to related discussions in [20, 23].

and the exclusion—considering also negative annotations and dependencies between features—as follows:

$$extent(\mathsf{f}) = \{\mathsf{e} \mid (\mathsf{e}, \mathsf{f}) \in \mathsf{A}_{\Rightarrow}\}$$
$$exclusion(\mathsf{f}) = \{\mathsf{e} \mid (\mathsf{e}, \mathsf{f}) \in \mathsf{N}_{\Rightarrow}\} \cup \bigcup_{(\mathsf{g},\mathsf{f}) \in \mathsf{M}} extent(\mathsf{g})$$

That is, the extent of a feature includes the extent of all implied features and the exclusion of a feature contains the extent of all mutually exclusive features. All recommendation mechanisms use these definitions of *extent* and *exclusion*; hence, they automatically reason about negative annotations and dependencies between features as well. All code elements that belong neither to $extent(\mathsf{f})$ nor to $exclusion(\mathsf{f})$ are undecided yet and are candidates for further mining of f.[4]

We use the definitions in the remainder of this section to illustrate each recommendation mechanism. In particular, we model prioritized recommendations as a set of weighted associations of elements to features $recommend \subseteq \mathsf{E} \times \mathsf{F} \times [0, 1]$.

3.2 Type System

The type system is our key recommendation mechanism and 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.

In previous work, we designed a variability-aware type system, which can type check an entire software product line in a single pass [22]. That is, instead of generating all (potentially billions of) variants in isolation, we designed an algorithm to check annotations against feature dependencies.

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 to method declaration, from variable access to variable declaration, and from type reference to type declaration. We look up references using the relationships R in our model (also at intraprocedural level). Between the identified pairs, we compare the annotated features. For example, if a method declaration is annotated with feature f whereas the corresponding method invocation is not annotated, the type system can issue an error message, because a variant without feature f will result in a dangling method invocation.[5] To be precise, we also need to look at the relationship between the involved features (domain knowledge). If, in our previous example, feature f was mandatory and included in all variants, we would not issue a type error; also if method declaration and invocation are annotated by different features, we do not issue a type error if the invocation's feature implies the declaration's feature. Also more complex relationships can be checked efficiently with the help of SAT solvers or similar tools [4, 22, 27, 36]. Similar to method invocations, we provide checks for field access, type references, local variables, and many other constructs.

Already when first experimenting with early versions of the type system over four years ago, we found using type errors for variability mining almost obvious. When

[4] In principle, inconsistent annotations are possible. For example, $extent(\mathsf{f})$ and $exclusion(\mathsf{f})$ 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 feature model). Our tool could issue a warning in case that happens and a developer has to fix the annotations manually.

[5] In fact, type checking is more complicated when language features such as inheritance, method overriding, method overloading, and parameters are involved. For such cases, we adjusted the type system's lookup functions. For details, we refer the interested reader to our formal discussions in [22].

annotating a code fragment, say method *lock* in Fig. 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.

We have already implemented such variability-aware type system for Java in prior work (and, for a subset, formally proved that it ensures well-typedness for all variants of the product line) [22]. For variability-mining, we reimplemented these checks as recommendation mechanism.

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 later.

To summarize our findings formally, the type checker can be seen as a function that takes the program elements E, annotations A, and variability model VM, and produces a set of type error messages of the form (e, f), to which we assign priority 1:

$$recommend_{TS} = typeerrors(\mathsf{E}, \mathsf{A}, VM) \times \{1\}$$

3.3 Topology Analysis

Next, we adopt Robillard's topology analysis [31] and adjust it for the product-line setting (fine granularity, domain knowledge). The basic idea is to follow the system-dependency graph 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 Fig. 1, when the method is mostly invoked by annotated statements (reference relationship); it recommends a local-variable declaration such as *l* in Fig. 1, when the variable is only assigned from annotated code elements (usage relationship); and it recommends an entire class, when the class contains mostly annotated children (containment relationship).

We do not list the specific ranking algorithm to calculate a recommendation's priority $weight_{TA}$ here, because we stay close to Robillard's algorithms. We adapt it only for the product-line setting: First, we determine relationships at all levels of granularity (i.e., down to the level of statements and local variables), whereas Robillard considers methods and fields only. Second, we consider relationships between features (domain knowledge, if available) by using the entire extent of a feature (cf. Sec. 3.1, includes annotations of implied features). 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 $exclusion(\mathtt{f})$ and subtract the result from the priority regarding the extent:

$$recommend_{TA} = \big\{(\mathsf{e}, \mathtt{f}, w) \,|\mathsf{e} \in neighbors(extent(\mathtt{f})),\\ w = weight_{TA}(\mathsf{e}, \mathtt{f}, extent(\mathtt{f})) - weight_{TA}(\mathsf{e}, \mathtt{f}, exclusion(\mathtt{f}))\big\}$$

3.4 Text Comparison

Finally, we use text comparison to derive recommendations between declarations (method, field, local-variable, and type declarations; a subset of all code elements), which are not restricted to neighboring elements as type system and topology analysis are. The general idea is to tokenize declaration names [7] and to calculate the importance of each substring regarding the feature's *vocabulary*. The vocabulary of a feature consists of all tokens in $extent(\mathtt{f})$. Intuitively, if many annotated declarations contain the substring "lock" (and this substring does not occur often in $exclusion(\mathtt{f})$), 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 number of occurrences of substrings in declarations in $extent(\mathtt{f})$ and subtract the relative number of occurrences in $exclusion(\mathtt{f})$. That is, negative annotations give negative weights to words that belong to unrelated features. Note that by using $extent(\mathtt{f})$ and $exclusion(\mathtt{f})$, we again consider domain knowledge (if available); for example, names used in mutually exclusive features influence the priority of recommendations.

We implemented an own mechanisms, because it was sufficient to add a simple recommendation mechanism. 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 cover a feature's vocabulary more accurately.

$$recommend_{TC} = \\ \{(\mathsf{e}, \mathtt{f}, weight_{TC}(\mathsf{e}, vocb(extent(\mathtt{f})), vocb(exclusion(\mathtt{f}))))\}$$

3.5 Putting the Pieces Together

For each code element, we derive a recommendation priority by merging the priorities of all three recommendation mechanisms. Following Robillard [31], we use the operator $w_a \uplus w_b = w_a + w_b - w_a \cdot w_b$ 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 $[0, 1]$). The overall priority is calculated as $w_{TS} \uplus w_{TA} \uplus w_{TC}$.

The three techniques are complementary, as we can illustrate on our initial stack example in Fig. 1. The type system finds many code fragments, which are critical by definition, because they must be fixed to achieve consistency. In our example, the type system recommends the invocations of method *lock* in Lines 7 and 12 once the corresponding method declaration (Lines 17ff) is annotated; the invocation would also be identified by the topology-analysis mechanism and text comparison, but with a lower priority. 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. In this case, also text comparison would fail (at least without additional ontologies), because it would not detect the semantic similarity between *transaction* and *locking*. Nevertheless, topology analysis would provide a rec-

ommendation. Finally, neither type system nor topology analysis would recommend the method declaration *getLockVersion* that is never called from within the implementation; here, text comparison can provide additional recommendations. This example illustrates the synergies of combining the three complementary recommendation mechanisms. Our tool is extensible; we could easily integrate additional recommendation mechanisms.

4 Evaluation

Our goal is to evaluate whether our recommendations guide developers to consider relevant code fragments.[6]

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 [20]. LEADT reuses CIDE's infrastructure for variability modeling and reasoning about dependencies, for the mapping between features and code fragments, and for rewrite facilities, once code is annotated. 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 Sec. 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 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 a reasonable stop criterion below.
4. Rewriting the annotated code (optional), possibly using CIDE's facilities for automated exports into conditional compilation and feature modules [21].

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. LEADT always provides recommendations for the current context and updates them on the fly.

4.1 Case Studies

Before quantitatively evaluating the quality of LEADT's recommendations, we report experience from two case studies. The case studies serve two purposes: (a) They illustrate how developers interact with LEADT in practice and (b) they informally explore benefits and limitations of variability mining. Even though we attempt to take

[6] 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 not be fair.

a neutral approach, even with an independent developer in one case study, our case studies provide anecdotal insights and are not suited or meant as objective generalizable evaluation.

4.1.1 HyperSQL

As first case study, we asked a domain expert to identify features in HyperSQL (a.k.a. HSQLDB; not developed as product line).[7] After a short end-user introduction to LEADT, we let him proceed freely, but recorded the screen during the feature mining process, recorded a think-aloud protocol, and conduced a subsequent interview with the expert.

We selected HyperSQL as target, a fast open-source relational database engine implemented in 160 000 lines of Java code. HyperSQL has not been developed as product line, but typical scenarios, such as embedding it into other applications (among others in OpenOffice 3.2), make a product-line approach reasonable, so that smaller, faster, and more specialized variants can be generated.

We recruited a PhD student from the database research group of the University of Magdeburg as expert, who has good knowledge on database architectures and who has recently analyzed HyperSQL and located some product-line features for an unrelated research project. The expert has actually already located three features manually before and is, hence, familiar with the challenges involved. According to our interview, he investigated the source code mostly following static dependencies with Eclipse' call-graph function and using the global search function. He had no background in concern-location approaches nor did he know the mechanisms of our feature-mining tool. For the case study, we briefly explained him how to use the feature-mining tool, but left open whether and how he actually used it.

Features. As target features, we selected *Profiling*, *Logging*, and *Text Tables* (without dependencies), which are always included in the implementation of HyperSQL. We use the two debugging features *Profiling* and *Logging* because the expert has already located these features previously, so they make a good warm-up task before locating a new feature. It makes sense to make both features configurable in a product line, because they are rarely needed in practice and variants without them are actually 1.4 % faster according to HyperSQL's default benchmark (ex-post analysis). Feature *Text Tables* comprises a substantial subset of the implementation, which allows querying and modifying tables stored as comma-separated-values text files (CSV). We selected this feature, because it is prominently described in the end-user documentation and rather uncommon for database systems. It was originally developed independently and later incorporated into HyperSQL. Locating *Text Tables* corresponds to a scenario in that a feature was added on request, but the company later decides to sell this feature only to selected customers or remove it from the main line to reduce byte-code size (ex-post, we measured a 3 % reduction of byte-code size).

Results. In Table 1, we summarize the results of mining all three features. Overall, the domain expert needed about four hours to locate all three features. It becomes visible that all features are scattered and mostly fine-grained in nature.[8] For feature

[7] `http://hsqldb.org/`; version 1.8.0.10.

[8] The number of methods includes only methods that are independently annotated, not methods that are part of an annotated container construct. The same applies for fields, statements, and parameters.

		Profiling	Logging	Text Tables
Feature Size and Scattering	Code Fragments	39	105	164
	Lines of Code	248	388	2819
	Files	7	16	26
	Packages	3	4	5
Granularity	Classes	1	1	6
	Fields	4	5	40
	Methods	1	9	25
	Imports	5	12	1
	Statements	28	78	89
	Parameters	0	0	3
Manual Rewrites	Parameters	0	0	10
Mining Effort	(in min)	25	40	180

Table 1: Variability Mining in HyperSQL

Text Tables, the expert had to rewrite 10 small code fragments, in which only parts of a statement belong to a feature, such as *"return isCached || isText;"*. The observed fine granularity of feature implementations is in line with our experience from other product lines [20, 23].

All three located features were *consistent*. We compiled and ran all eight variants successfully. Furthermore, the expert was confident they were complete as well.

Mining Process and Experience. For features *Profiling* and *Logging*, the expert each knew one class that he could use as starting point *(StopWatch* and *SimpleLog)*. For *Text Tables*, he globally searched for configuration parameters mentioned in the user documentation ("textdb.fs" and seven similar ones). Afterward, he usually followed recommendations provided by LEADT ordered by priority.

Despite generally following LEADT's recommendations, looking beyond the scope of an individual recommendation was quite natural. For example, when the tool recommended a single statement, often he looked also at similar statements in the same method; whenever patterns emerged (e.g., "in this file all references to a field *appLog* belong to feature *Logging*"), he switched to text search ("find next") within the same file; in one case he manually looked for the declaration of a local variable many lines before its use (it was used inside a recommended statement). Deviations from recommendations mostly occurred within a single file. After manually processing that file, he returned to the recommendations to check whether he missed something or with an expression like "this file seems done, what else". That is, *he quickly trusted our tool's recommendations*.

In most cases, the expert could quickly decide whether a recommended code fragment belonged to the target feature. When the decision was not obvious from the name, from previous annotations, or from local context, he looked at a larger context, at the call hierarchy (to confirm a non-obvious recommendation), and, in *Text Tables*, at the end-user documentation and JavaDoc comments in the source code. In three cases, the expert postponed the decision about a recommendation, but returned to them later in the process, when the decision became obvious because the recommendation was supported by additional feature code that was located in between.

Feature	Annotations	Lines of Code
Logging	1 245	2 067
State	529	21 963
Activity	171	7 963
Critics	359	37 649

Table 2: Mined features in ArgoUML

The expert followed most recommendations and annotated the according code fragment (the recommendations felt valuable and precise). Nevertheless, also negative annotations helped; when adding a negative annotation, in two cases, many similar and also incorrect recommendations disappeared from high ranks in the recommendation list.

There was no clear line when to stop the mining process. For the first two features, the expert stopped locating more code intuitively when several recommendations in a row were incorrect and priorities dropped below 0.6. For both features, he skimmed the next screens of recommendations (about 40–60 recommendations) and carried out some random checks, but found no more additional code. For *Text Tables*, he continued for some time with textual search, looked at the end-user documentation again, and eventually found two more code fragments that the tool did not recommend with sufficiently high priority.

Limitations. Finally, the case study provided several insights for future work. First, the expert strongly preferred to process recommendations locally, instead of jumping between different code fragments with the highest priority. This raises interesting questions for future work whether we can include the distance between recommendations to order recommendations with similar priority. Second, the expert suggested providing an explicit *waiting list*, where he could store recommendations that he would not want to decide right away.

4.1.2 ArgoUML

As second case study, we report from our experience in mining features in ArgoUML[9] (305 000 lines of Java code). We targeted the same features *Logging*, *State Diagram*, *Activity Diagram*, and *Design Critics*, as done in a previous decomposition (first manual using conditional compilation [11], subsequently repeated semiautomatically [38]). The features are heavily scattered and cover large portions of the application. Table 2 summarizes the size, measured after our decomposition. We attempted to locate the same features, without looking at the previous decompositions (we only investigated the previous decomposition ex-post) and without any knowledge about ArgoUML's implementation.

Although ArgoUML may not be a typical candidate for adopting a product-line approach (there is little incentive to justify the overhead of a product-line mechanism for this kind of desktop applications), in this study, we could explore LEADT in a large code base. Since we had no domain knowledge, we interactively worked with LEADT, made errors and reverted changes. We could furthermore explore what degree of automation is possible and desirable. In the following, we report our main observations.

[9] `http://argouml.tigris.org/`; version 0.28; SVN revision 16938.

Mining Process and Experience. For all four features, seeds were obvious. The diagram and critics features correspond to larger subsystems, so we could start with entire packages as seeds. For feature *Logging*, the log4j framework was an obvious seed.

First, we mined the feature *Logging*. It turned out that logging follows only trivial pattern (in many files: import of the *log4j* framework, declare and initialize a field, and invoke a logging method several times). Our recommendation mechanism is well capable of finding all corresponding locations (actually, even each recommendation mechanism in isolation would have performed similarly). Actually, this process was so simple that LEADT adds only little value to a manual approach, beyond the common bookkeeping (ensuring that we do not forget any annotations). On the other hand, the process was so repetitive (over 1200 one-line annotations), so more automation would be desirable: Instead of just recommending locations, in this case, an "annotate all" mechanism for a set of recommendations would have reduced the developer's effort.

Second, we located feature *State Diagram*, which was harder than initially expected. The problem is that, without knowing the internals of ArgoUML, it was not always obvious how to decide which code fragments actually belong to the feature. For example, LEADT often recommended code fragments that appeared to be general infrastructure for several kinds of diagrams. On closer investigation, we found that the code fragment was currently only used by code from feature *State Diagram*. Depending on our goals (for example, minimal binary size or future extensibility), we could justify both decisions: annotating and not annotating these code fragments.

Although large parts of the feature were already grouped in two packages (which provided obvious seeds), there was plenty of scattered code for hooking the diagrams into the user interface and so forth. In contrast to feature *Logging*, there was hardly any obvious automation potential. For each recommendation, we needed to investigate the code fragment to decide whether to add an annotation. Often, after manual code inspection, we added annotations larger than the initial recommendation (e.g., annotate the entire file, even though just a statement was recommended).

Only after several incorrect annotations, we found out that feature *Activity Diagram* is actually not orthogonal to *State Diagram* but reuses its infrastructure. Once we realized that problem, we changed several incorrect annotations from *State Diagram* to *Activity Diagram* and added a dependency *Activity Diagram* $\rightarrow$ *State Diagram* before continuing with both features (an alternative design decision would have been to add a new feature providing common functionality to both features). Due to LEADT's design, we could easily change annotations and change the feature model during the mining process.

Finally, we located feature *Design Critics*. This feature was again rather straightforward to extract as it already presented an entire subsystem. We could start with several entire packages containing the main functionality and then followed recommendations throughout the source code to fix (a quite large amount of) remaining scattered code, typically invocations to code within the packages and additional classes for the graphical user interface. In most cases, it was obvious from the code structure or comments, which code fragments belong to a feature. Like in feature *Logging*, there were stable patterns that would have allowed some degree of automation.

In general, in our experience with ArgoUML, LEADT significantly contributes to the mining process. Although we often proceeded manually from some points, we always returned to LEADT's recommendations as main guidance through the mining process. The provided granularity fits well to the task.

```
TodoTokenTable
OffenderXMLHelper
XmiFilePersister.todoString
MemberList.todoList
MemberList.setTodoList
ApplicationVersion.getManualForCritic
ShortcutMgr.ACTION_OPEN_CRITICS
GenericArgoMenuBar.critique
```

Figure 2: Some additional code elements in ArgoUML for feature *Design Critics*.

Comparison. Comparing our decomposition with the previous manual one [11] is difficult: We can report differences but mostly cannot soundly judge which decomposition is better; too many influence factors may affect that decision. For feature *Logging*, our decomposition was almost identical to the previous one, which is not surprising, given the simple patterns. In the other features, the decompositions were similar, but we located additional code fragments.

The differences are clearest in feature *Design Critics*. We identified a strict superset of feature-code locations compared to the manual decomposition.[10] In addition to the code fragments identified in the original decomposition, we located 37 additional code fragments (together 637 lines of code, 6 classes, 9 fields, 12 methods, 15 statement sequences, and 5 parameters). Several of these code fragments appeared quite apparent, because they explicitly refer to "critics" or "todo" (part of the feature); we list some examples in Fig. 2.

For other code fragments in *Design Critics* (and also both diagram features), it was less obvious where to draw the line. For example, large parts of the OCL infrastructure,[11] such as class *OclInterpreter*, are used only in the context of *Design Critics*, but were not entirely extracted. Again different decisions can be justified and we decided annotated more code fragments.

Also for both *State Diagram* and *Activity Diagram*, we located a superset with more feature code than the previous decomposition. For example, we located several graphical representations of objects and critics that occur only in that diagram type (e.g. *CallStateNotation*, *TransitionNotation*, and *CrTooManyStates*), which were not annotated before.

All additionally located code was "dead code" when the target feature was not selected. It was infrastructure code, currently only used by the target feature (such as OCL for design critics). We conjecture the following reason for most differences: In the manual approach, the authors tried to compile and run different variants after decomposition. In that process, they would detect type errors (as does our type-system recommendation mechanism). However, as explained in Sec. 3.5, the type system alone is not sufficient. Especially it misses dead code; whereas LEADT's topology-analysis recommendation mechanism is effective at detecting such dead code as feature code.

Limitations. On a technical side, we noted that LEADT's implementation does scale to large programs, but struggles with large features. When annotating many code

[10] In fact, there were two instances of a simple bug: Two actions import an incorrect version of the *Translator* class (equivalent behavior); however, instead of changing the import, authors of the original decomposition removed translation altogether by annotating text initialization of both actions.

[11] Object Constraint Language, a declarative language for describing rules that apply to UML models.

fragments (such as all code in several packages) the topology-analysis and the text-comparison recommendation mechanisms require substantial time to update their recommendation. In ArgoUML, LEADT needed several seconds or even minutes to update the recommendation list after each annotation. Although this problem is caused partially by our decision for fine granularity, we believe that this is mostly an implementation issue that can be solved by simple caching strategies.

4.1.3 Summary and Perspective

Both case studies show that LEADT can guide developers effectively in the variability-mining process in large and realistic code bases. In both cases, we located features in legacy code that was not developed as product line. The HyperSQL case study shows that independent developers actually understand the tool and appreciate the support, although it is not used all the time. In the ArgoUML case study, we replicated a previous decomposition and found that LEADT is particularly effective at finding dead code that was missed previously.

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. Our mining tool only recommends potential locations of feature code, whereas a developer has to decide whether this code fragment belongs to a feature. 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 do not attempt to generalize the case-study experience, but concentrate on a quantitative evaluation in a controlled setting next.

4.2 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 the overall amount of feature code in the original implementation (measured in lines of code). *Precision* is the percentage of correct recommendations compared to the overall number of recommendations of the tool.

4.2.1 Study Setup and Measurement

The critical part of an experiment measuring recall and precision is the benchmark—the assumed correct mapping between code fragments and features. An incorrect benchmark would lead to incorrect results for both recall and precision.

To combat experimenter's bias, we do not design the benchmark ourselves or rely on domain experts that might be influenced by the experimental setting. Instead, to find benchmarks, 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-locate all features.

Our strategies exclude experimenter bias, but limit us in our selection of benchmarks. We cannot simply use any large scale Java application, as we did when locating new features in HyperSQL (see case studies). Similarly, we cannot use any previous case

studies that were created with an early version of the type system in CIDE, such as BerkeleyDB [20]. The resulting trade-off between internal validity (excluding bias) and external validity (many and large studies) is common for decisions in experimental design; we decided to emphasize internal validity.

After selecting the benchmarks, the evaluation proceeds as follows: First, we create a variability model, reusing the names and dependencies from the benchmark.[12] Second, we add seeds for each feature (see below). Third, we start the mining 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 largest code fragment); if, according to the benchmark, the recommended code fragment belongs to the feature, we add an annotation, otherwise, we add a negative annotation.[13] We iteratively repeat this process until there are no further recommendations or until we reach some stop criteria (see below). After stopping, we determine recall by comparing the resulting annotations with the benchmark and precision by comparing the numbers of correct and incorrect recommendations. Finally, we continue the process with the next feature. Since we exclude all human influence, measurement is repeatable and we automated it.

There are different strategies to determine seeds. We use a conservative strategy without experimenter bias based on an existing tool—the information retrieval engine of FLAT^3 [32] (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 correct result (a field, a method, or all relevant statements inside a found method). We discuss the influence of different or more seeds in Sec. 4.2.4.

Deciding when to stop the mining process for a feature (stop criteria) is difficult, as the developer cannot compare against a benchmark. 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 Sec. 4.2.4.

4.2.2 Benchmarks

We selected four different benchmarks 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 product line, but has been manually decomposed into features at least three times [16, 24, 38]. Since all previous decomposition agree almost perfectly on the extent of each feature and

[12] We assume that a domain expert would construct a similar model. In addition, we separately evaluate the impact of missing dependencies and different seeds below.

[13] Actually, mirroring human behavior (experienced in HyperSQL and others), when a specific recommendation is correct, we also look at the direct surrounding code elements and annotate the largest possible connected fragment of feature code. For example, if the tool correctly recommends a statement and the benchmark 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 3: Feature characteristics and mining results.

since Prevayler was *not* developed as product line, Prevayler is a perfect benchmark for variability mining. 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 $\texttt{Censor} \rightarrow \texttt{Snapshot}$. We also investigated several manual decompositions of other projects, looking for further high quality benchmarks there, but non of them 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) [15], MobileMedia contains six features, *Photo*, *Music*, *SMS Transfer*, *Copy Media*, *Favourites*, and *Sorting*, with the following dependencies: $\texttt{Photo} \vee \texttt{Music}$ and $\texttt{SMSTransfer} \rightarrow \texttt{Photo}$.[14] We added a feature *Media Transfer* and the dependency $\texttt{MediaTransfer} \leftrightarrow (\texttt{SMSTransfer} \vee \texttt{CopyMedia})$ to cover the code that is common to the two transfer features (which is implemented in the original implementation with *#ifdef SMS || Copy*). Unfortunately, FLAT^3 would not find any relevant feature code for *Media Transfer*, but thanks to domain knowledge, we could mine it without seeds (cf. Sec. 4.2.4). Despite being a medium-sized academic case study, MobileMedia is a suitable benchmark, because its Java ME code is well maintained and peer reviewed [15]. The analyzed version was implemented with conditional compilation; so, we derived a base version by running the preprocessor with all features selected.

- **Lampiro.** The open-source instant-messaging client Lampiro, developed by Bluendo s.r.l. with 44 584 lines of Java ME code (147 files),[15] 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).

- **Sudoku.** The small *Sudoku* implementation (1975 lines of Java code, 26 files), result of a student project at the University of Passau, Germany, contains five features: *States*, *Undo*, *Solver*, *Generator*, *Variable Size* (we exclude a sixth feature *Color* because of its small size). Although the project is implemented with a distinct code unit for each feature, which can be composed using FeatureHouse [1] (similar to aspect weaving),[16] we reconstructed a common code base and corresponding annotations. Despite the small size, *Sudoku* is interesting, because most of the features incrementally extend each other, hence the following dependencies: $\texttt{Generator} \rightarrow \texttt{Solver}$, $\texttt{Solver} \rightarrow \texttt{Undo}$, and $\texttt{Undo} \rightarrow \texttt{States}$.

In Table 3, 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*)

[14] Source code: `http://mobilemedia.cvs.sf.net`, version 6_OO, last revision Oct. 2009. We use the feature names published in [15], 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 $\texttt{SMSTransfer} \rightarrow \texttt{Photo}$ to the variability model, which we detected in prior work [22].

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

[16] Code available as part of the FeatureHouse case studies `http://fosd.net/fh`.

is quite common, but unproblematic; we simply need to locate such code fragments for each feature. All benchmarks are available (e.g., for replication or comparison) in LEADT's repository.

4.2.3 Variability-Mining Results

In Table 3, 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 benchmark (academic vs. industrial, single application vs. 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). 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 using the same string literals (text comparison currently only compares names of 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 are 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). In addition, keep in mind that our stop criteria 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 criteria (not considering the last ten incorrect recommendations would improve the overall average precision from 42 to 76 %).

4.2.4 Further Measures

Beyond our default setting, we investigated the influence of several parameters more closely. For brevity, we concentrate on the main insights.

Influence of domain knowledge. Although not directly visible from Table 3, domain knowledge about dependencies between features can have a significant impact on the results of the mining process. The influence becomes apparent when mining features with dependencies in isolation or in different orders. We selected the order in Table 3 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 serve as additional 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.

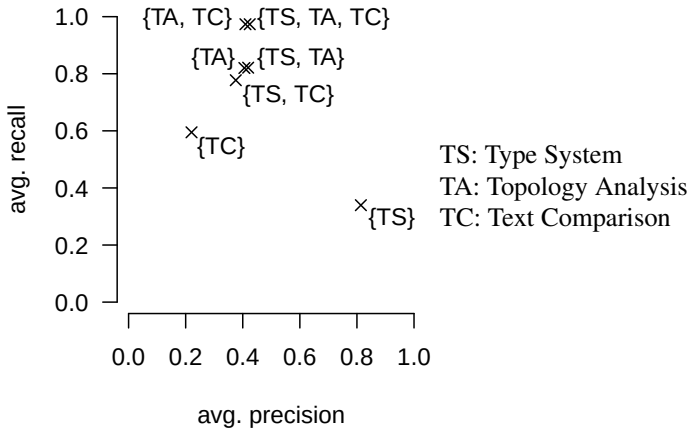

Figure 3: Combining recommendation mechanisms.

Importance of the recommendation mechanisms. The recommendation mechanisms contribute to different degrees to the results. We explored different combinations of the recommendation mechanisms and plot the resulting average recall and precision over all case studies in Fig. 3. Especially type system and text comparison are not effective on their own. As described in Sec. 3.5, the mechanisms are complementary—combining them improves performance.

The results may seem as if the type system, at least in our setting, contributes little compared to topology analysis. Nevertheless, there is a significant difference in the quality of recommendations. All 101 recommendations of the type system (19 % of all recommendations) lead to new annotations, whereas the other recommendations have a much lower precision (even of 221 other recommendations with the hightest priority 1, only 64 % were correct) . The only reason the type system does not always score 100 % precision in isolation is that it may recommend code from dependant features (a developer would probably recognize the problem and annotate the code with the correct feature).

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 3. Also handpicking larger seeds, as a domain expert might do, leads to a similar recall. This shows that the recommendation mechanisms are quite efficient finding connected fragments of feature code, almost independent of where the mechanisms starts. Huge seeds as in the ArgoUML case study can help, but are not necessary.

Stop criteria. Finally, we have a closer look at the stop criteria. Note that we selected our stop criteria before our evaluation; although we could determine a perfect criteria ex-post, we could not generalize such criteria. In Fig. 4, we plot the average recall and precision for mining all 19 features with different stop criteria. We can observe that up

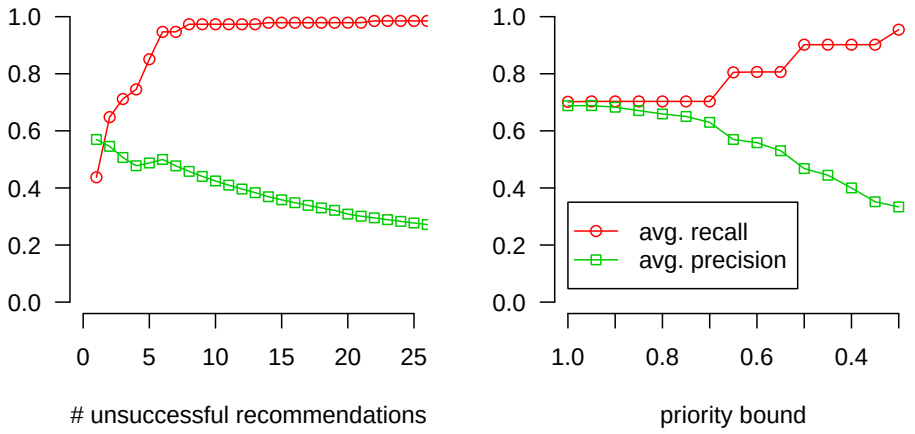

Figure 4: Alternative Stop Criteria.

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 criteria.

4.3 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.

In our quantitative evaluation, we attempted to maximize internal validity and exclude bias as far as possible by using neutral benchmarks. Regarding external validity, the selection of four rather small benchmarks with few features each still does not allow generalizing to other software systems. Furthermore, the selection of existing product lines as benchmark could introduce new bias: Potentially, because the system was already implemented as 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.

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* [3, 5, 34, 35]. 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 [13]. 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.

There is a vast amount of research on (semi-)automatic techniques to locate concerns, features, or bugs in source code, known as concept assignment [6], concern location [14], feature location [32], impact analysis [29], 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, such as 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; however, they typically refer to finer-grained scattered implementations (typically they trace individual methods instead of entire components) and are used for an one-time task only.

Many different techniques for concern location exist: there are static [6, 29, 31] as well as dynamic [10] and hybrid [14, 32] techniques, and techniques that employ textual similarity [14, 32] as well as techniques that analyze static dependencies or call graphs [6, 14, 29, 31] and program traces [10, 14, 32, 34]. For a comprehensive overview, see [10] and [31]. Many of them complement our approach 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* [31] 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 Sec. 3.3, we extended Suade's mechanism with domain knowledge and use a more fine-grained model including 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 [29]. 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 additionally adds a concept called *prune-dependency analysis* to find the complete extent of a concern [14]. 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 [17]. Gilligan is tailored specifically for reuse decisions, to locate and copy code excerpts from legacy code. In this scenario, a key decision is when *not* to follow such dependencies and replace them with stubs in the extracted code—whereas in product lines, located features are reused within a single implementation. 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. However, our type system is more fine-grained and additionally considers domain knowledge about relationships between features.

Finally, beyond traditional concern-location techniques, *CIDE+* is the closest to our variability-mining concept [38]. 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 [20]). In contrast to our approach, they solely use a type-system-like mechanism, similar to Cerberus' prune dependency analysis [14], 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 individual recommendations to developers. In Sec. 4.2.4, we have demonstrated the benefit of domain knowledge and 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.

6 Conclusion

Software product lines are a 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 for the need of software product lines (consistency indicator, fine granularity, domain knowledge). We have demonstrated that variability mining can effectively direct a developer's attention to relevant feature code. In future work, we intend to explore synergies with further recommendation mechanisms, especially ones based on dynamic traces and on deltas between existing variants.

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 and Sven Apel for comments on a prior draft of this paper. Dreiling's work was supported by the Metop Research Institute. Käster and Ostermann's work is supported by ERC grant #203099.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2011. accepted for publication.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse Engineering Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE)*. 2011. to appear.

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

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

[36] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. 2007.

[37] 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.

[38] M. T. Valente, V. Borges, and L. Passos. A Semi-Automatic Approach for Extracting Software Product Lines. *IEEE Trans. Softw. Eng. (TSE)*, 2011. to appear.

[39] 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.