PEoPL: Projectional Editing of Product Lines

Benjamin Behringer
University of Luxembourg, Luxembourg
htw saar, Germany

Jochen Palz
htw saar,
Germany

Thorsten Berger
Chalmers | University of Gothenburg,
Sweden

***Abstract*—The features of a software product line—a portfolio of system variants—can be realized using various implementation techniques (a.k.a., variability mechanisms). Each technique represents the software artifacts of features differently, typically classified into annotative (e.g., C preprocessor) and modular representations (e.g., feature modules), each with distinct advantages and disadvantages. Annotative representations are easy to realize, but annotations clutter source code and hinder program comprehension. Modular representations support comprehension, but are difficult to realize. Most importantly, to engineer feature artifacts, developers need to choose one representation and adhere to it for evolving and maintaining the same artifacts.**

We present PEoPL, an approach to combine the advantages of annotative and modular representations. When engineering a feature artifact, developers can choose the most-suited representation and even use different representations in parallel. PEoPL relies on separating a product line into an internal and external representation, the latter by providing editable projections used by the developers. We contribute a programming-language-independent internal representation of variability, five editable projections reflecting different variability representations, a supporting IDE, and a tailoring of PEoPL to Java. We evaluate PEoPL's expressiveness, scalability, and flexibility in eight Java-based product lines, finding that all can be realized, that projections are feasible, and that variant computation is fast (<45ms on average for our largest subject Berkeley DB).

I. Introduction

A software product line (SPL) is a portfolio of systems engineered in a specific application domain, such as telecommunication, automotive or industrial automation [1], [2]. The individual systems of an SPL, called *variants* or *products*, share commonalities and variabilities. As such, constructing an SPL amounts to engineering common and variable software artifacts, each of which realizing one or several *features* [3], [4]. Individual variants are derived from the SPL in an automated process by selecting the desired features of the variant.

Many implementation techniques—so-called *variability mechanisms*—have emerged for engineering SPLs, such as variability annotations [5]–[9], templates [7], deltas [10]–[12] or feature modules [13]–[16]. These techniques represent features differently, typically classified into *annotative* and *modular* approaches, each having their own advantages and disadvantages. Annotative approaches—e.g., the C preprocessor (CPP)—represent all feature artifacts directly in the codebase by wrapping them with annotations. Such annotations are easy to apply, but challenge program comprehension [17] by obscuring the structure and data-flows of source code [6]—hampering editing experience and negatively impacting maintenance and evolution [18]. Moreover, developers always see all possible variants, many of which might not be relevant for the current engineering activity. Modular approaches—e.g., AHEAD [14], and FeatureHouse [16]—represent all of a feature's artifacts in one module. They facilitate a clear structure of the system and allow engineering features without being distracted by irrelevant ones. Yet, decomposing a system into modules is challenging, since it requires finding the right decomposition strategy, and since creating modules imposes substantial overhead.

Although these representations are complementary [19]–[21], existing SPL engineering approaches typically focus on one representation. Most importantly, these approaches force developers to choose one representation for developing a feature artifact and to adhere to it for evolving and maintaining this artifact. While refactorings were proposed for switching between annotative and modular representations [19], such refactorings are heavyweight and do not allow to quickly switch the representation for a feature artifact. Ideally, developers could exploit the benefits of different representations on-demand and always choose the one that suits the current engineering activity.

We present the approach PEoPL (Projectional Editing of Product Lines) to realize this flexibility. It allows developers to flexibly choose the best-suited among very different representations of feature artifacts. Developers can also use various representations in parallel (side-by-side) for the same artifact. The core idea of PEoPL is to establish an internal representation of the SPL and separate it from the external representations that developers use. Fig. 1 illustrates this idea. Internally, the feature artifacts are uniformly represented in a variational abstract syntax tree (AST), whose nodes are annotated using concepts from our variability language CoreVar, which is programming-language-independent. The variational AST is manipulated using editing operations we conceived upon CoreVar. For meaningful variational ASTs, CoreVar can easily be tailored to specific programming languages. We provide such a tailoring for Java, declaring which of its language concepts are annotatable. Externally, this variational AST is represented

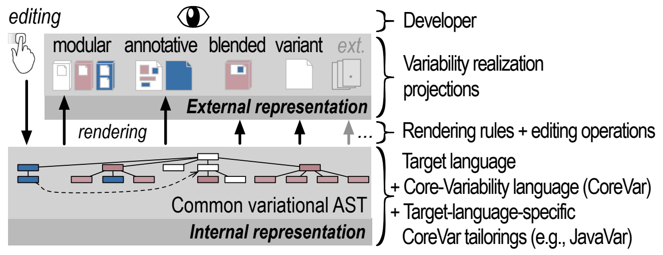

editing
modular
annotative
blended
variant
ext.
External representation
rendering
Common variational AST
Internal representation
Developer
Variability realization projections
Rendering rules + editing operations
Target language
+ Core-Variability language (CoreVar)
+ Target-language-specific CoreVar tailorings (e.g., JavaVar)

Figure 1. PEoPL separates internal and external variability representations

using different editable projections, which developers use to engineer feature artifacts. Any of their editing activities directly change the underlying AST, which immediately updates all projections. We conceive projections showing feature artifacts as (i) textual annotations (#IFDEF), (ii) visual annotations (colored bars), (iii) feature modules, (iv) annotations blended into feature modules, and (v) variants (i.e., hiding artifacts related to non-selected features).

We show the feasibility of our approach by realizing a complete IDE built upon the language workbench Jetbrains Meta Programming System (MPS). It realizes our projections and operations for engineering feature artifacts, while benefiting from common program-editing facilities (e.g., for Java) in MPS.

We evaluate PEoPL using eight Java SPLs. Our largest subject (and running example) is Berkeley DB, an SPL with 70kLOC including 42 features and 218 classes. PEoPL's expressiveness suffices to realize all SPLs without workarounds. The evaluation also shows that it is feasible and practical to conceive an internal representation that is projected into very different external representations. Furthermore, our approach scales: all projections can be rendered and edited without introducing significant latencies. For instance, variant editing is smooth, since computing a specific file variant (<1ms on average for all subjects) and calculating all AST nodes included in a variant is quick (<45ms for Berkeley DB on average).

We contribute the core variability language CoreVar, a tailoring of it to Java, five editable projections, an IDE realizing our approach, evaluation data for eight SPLs, and an online appendix [22] with a replication package and screencasts.

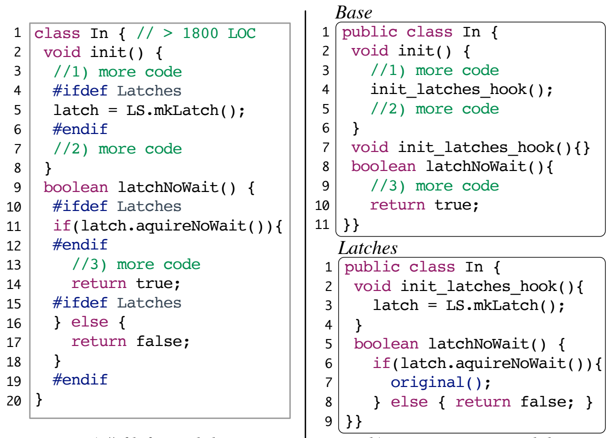
II. Motivation and Background

We briefly discuss contemporary approaches to engineer SPLs together with their advantages and disadvantages.

Annotative Representations. Mechanisms such as the CPP and CIDE [8] represent feature artifacts as annotation markers embedded into source code. The markers are either textual [7], [23] (e.g., #IFDEF) or visual [8], [24] (e.g., background colors), and have a Boolean expression over features. Variants are derived by removing the annotated code whose expression evaluates to *false* for a concrete selection of features.

Fig. 2a shows a CPP-based excerpt from Berkeley DB. #IFDEFs (e.g., Line 4) and #ENDIFs mark the beginning and the end of variable source parts belonging to a feature. Notice Lines 10 and 15 partially annotating the `if`-statement (i.e., not the body). Such annotations are called undisciplined. In Java, annotations on one or a sequence of entire classifiers (e.g., classes), members (e.g., method declarations) and statements are disciplined, all others are undisciplined. Since in practice around 16% of all annotations are undisciplined [25], approaches such as PEoPL need to support them.

Annotations are easy to incorporate and frequently used in practice [20], [26]. Artifacts can be annotated in an ad hoc fashion on a very fine-grained level (individual code lines or AST nodes). For instance, annotations can even wrap expressions or method parameters. In contrast, annotations are known to negatively impact program comprehension [6],



```
1  class In { // > 1800 LOC
2   void init() {
3    //1) more code
4    #ifdef Latches
5    latch = LS.mkLatch();
6    #endif
7    //2) more code
8   }
9   boolean latchNoWait() {
10   #ifdef Latches
11   if(latch.aquireNoWait()){
12   #endif
13     //3) more code
14     return true;
15   #ifdef Latches
16   } else {
17     return false;
18   }
19   #endif
20 }
```

a) #ifdef variability

Base

```
1  public class In {
2   void init() {
3     //1) more code
4     init_latches_hook();
5     //2) more code
6   }
7   void init_latches_hook(){}
8   boolean latchNoWait(){
9     //3) more code
10    return true;
11 }}
```

Latches

```
1 public class In {
2  void init_latches_hook(){
3    latch = LS.mkLatch();
4  }
5  boolean latchNoWait() {
6    if(latch.aquireNoWait()){
7      original();
8    } else { return false; }
9 }}
```

b) FeatureHouse modules

Figure 2. Berkeley DB feature *Latches* realized in CPP and FeatureHouse

[18], [27] as well as maintenance and evolution [17], [28]. They clutter source code and developers have to work with all variants at a time, since features cannot be edited in isolation.

Modular Representations. Mechanisms such as AHEAD [14], FeatureHouse [16], and DeltaJ [10], [12] represent the artifacts of a feature (and the interaction of features) in a cohesive unit called *composition unit*, *feature module*, *delta module*, or just *module*. The key idea is to start with a common base module and use other modules to introduce new structural elements, refine existing elements [14], [29], and even remove elements [10] step by step. Deriving a variant amounts to composing or applying the modules in a given ordering [10], [16].

Fig. 2b shows a FeatureHouse realization of the two Berkeley DB features: *Base* and *Latches*. The method `latchNoWait` is introduced by the feature *Base* (Lines 8–11) and refined by *Latches* (Lines 5–9). Note FeatureHouse's keyword `original` in Line 7 to call the original implementation of `latchNoWait` in *Base*. Refining the method `init` requires a hook method in the *Base*-code (Line 4), as modular approaches do not support fine-grained changes [8]. The hook's declaration is left empty in *Base* (Line 7), since it is overridden by *Latches* (Lines 2–4).

Modularity fosters comprehension [29]–[31]. A module can be edited without being distracted by irrelevant feature code [32]. For example, instead of searching for *Latches* code in around 1800 LOC (Fig. 2a), developers can just explore the respective feature module (Fig. 2b), which is less intellectually challenging [21], [33]. In contrast, the effort to create modules is typically high, since developers need to find the right decomposition strategy, which hinders adoption in practice. It is also difficult to realize fine-grained features, which might require boilerplate code (e.g., hook methods) or code clones.

Variant Representations. Typically realized upon an annotative representation, so-called *variation-control systems* or *variant editors* allow editing SPL variants in isolation. Some approaches fold [34] or hide [8], [24] feature code, others mimic the workflow of version-control systems by allowing to checkout variants, edit them, and commit the edited variants [35]–[37]. In Fig. 2a, for instance, we would hide a feature's annotations and code if irrelevant for the current code editing task (e.g., modifying code of the feature *Latches*).

Variant editors reduce editing complexity, which can positively impact efficiency (up to 40 % [38]). In contrast, they do not provide true modularity and easily face code-alignment issues (with respect to the hidden code). Furthermore, a checkout-edit-commit workflow imposes an overhead.

III. PEoPL Overview

We discuss PEoPL's key benefits and illustrate how developers can use it. We also provide an overview of its architecture.

Fig. 3 shows an excerpt of our running example Berkeley DB realized in PEoPL. The upper half of the figure shows the different projections we explain in this section. The lower part, illustrating the internal representation, is described in Sec. IV.

A. PEoPL's Benefits

The benefits of our approach can be summarized as follows.

First, PEoPL has a *uniform internal representation* (a variational AST defined by the CoreVar language), designed to support diverse external variability representations. Uniformity allows persisting variability in a consistent manner. Using different (internal) representations for feature modules and for annotations would break uniformity—for instance, when adding #IFDEFs into FeatureHouse modules.

Second, developers can *switch the external representation* of an artifact (e.g., class) on demand. PEoPL allows a fluent movement between external representations (e.g., of `DatabaseImpl` in Fig. 3) to enable developers to exploit the distinct advantages of different techniques for a given task.

Third, developers can observe and edit the same artifact *using different external representations in parallel* (by showing them side-by-side), enabling an even faster movement between representations. Moreover, it helps observing the impact of changes made in one representation to another in real-time (e.g., editing a feature module and a variant in parallel).

Fourth, PEoPL *mitigates typical shortcomings of modular representations*, imposed by granularity problems, and the lack of context information (as they are contained in external modules). By blending annotations into modules on demand, developers can realize fine-grained changes in feature modules (without breaking modularity and uniformity), and integrate context information from other modules on demand (e.g., accessible field and method declarations).

Fifth, our uniform internal representation enables *plugging new external representations into PEoPL* on demand. Thus, PEoPL can serve as a framework to evaluate representations.

B. PEoPL in Practice

We conceive, realize, and evaluate five external representations.

Textual Annotation Projection. Reflecting CPP's popularity, we provide a projection with CPP annotations, as shown in Fig. 3a. As most developers are familiar with CPP, yet not with PEoPL's other representations, starting by adding and exploring #IFDEFs is a useful option. Notice that PEoPL also supports various undisciplined annotations. Lines 4–8, 20, 24 show such undisciplined annotations on types, method parameters, and wrappers. The latter are program elements that wrap code blocks (body). In practice, it is sometimes necessary to only make the wrapper variable, but not its body (wrappee).

Visual Annotation Projection. When #IFDEFs clutter code and challenge comprehension, we can switch to visual annotations. The learning curve is low, since #IFDEF directives and visual annotations can be explored for the same feature artifact in parallel. Fig. 3c shows a projection of the class `DatabaseImpl` with annotations represented as colored bars, each related to a feature declared in Fig. 3b. For instance, light-gray bars relate to feature *Base*. Vertical bars are shown to the left of the program code and align with its indentation. Horizontal bars underline fine-grained feature artifacts within a line of code (e.g., method call parameters in Line 14) and partially annotated wrappers (e.g., the try/catch statement in Line 16). The ⊕-sign in Line 9 makes alternatives explicit.

Module Projection. Now, imagine we want to evolve the *Memory_Budget* feature or fix a bug in it. Obviously, it is beneficial to edit the class *DatabaseImpl* in isolation, and therefore switch from the annotative projection to the feature module implementing *Memory_Budget*. Fig. 3d shows the projection. Similar to AHEAD, the `refines` keywords indicate that the feature module *Memory_Budget* modifies the class `DatabaseImpl`, the inner class `PreloadProcessor`, and its method `processLSN` (Lines 2–8).

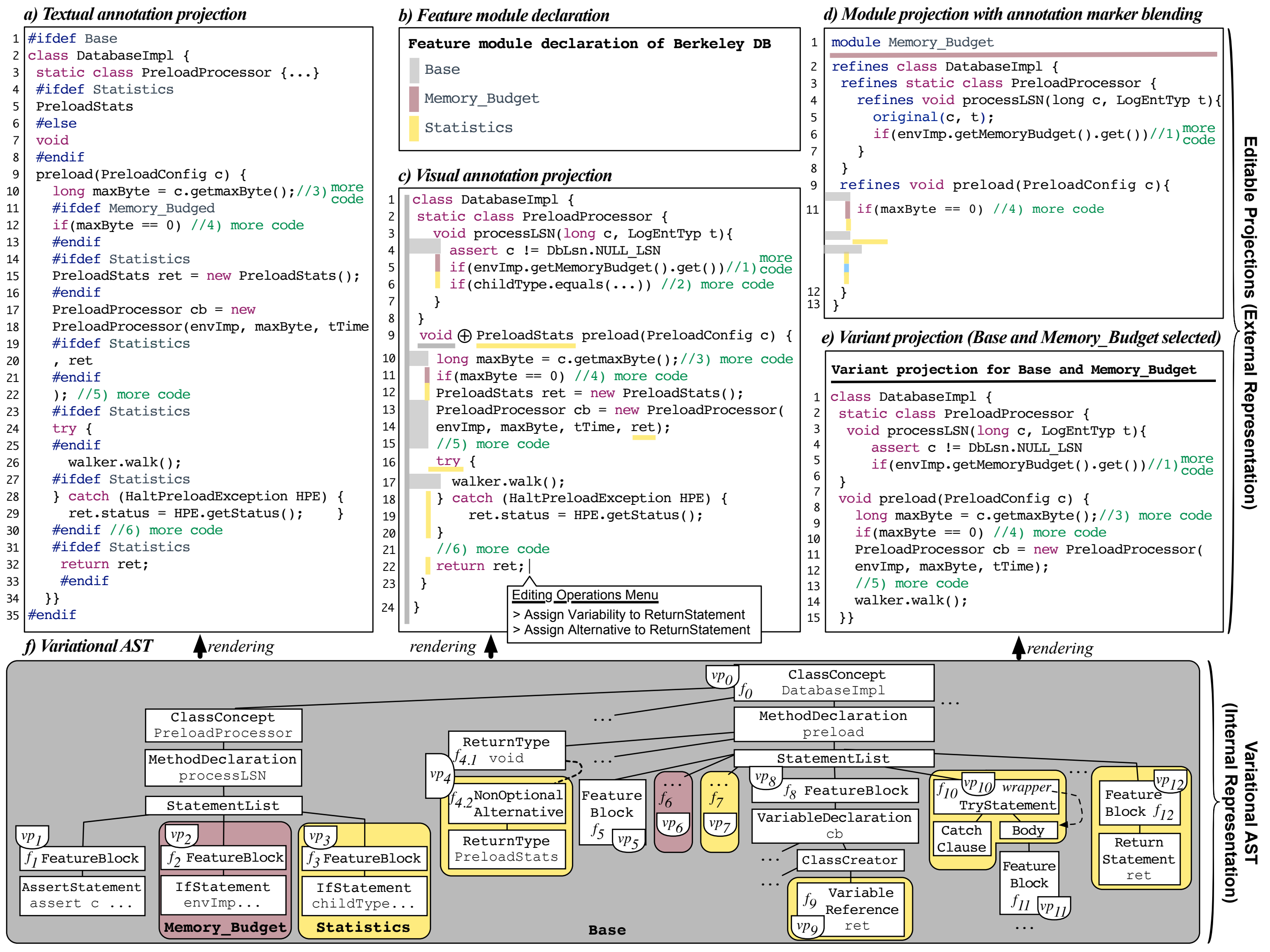
Blending Projections. Fig. 3c shows several fine-grained feature artifacts: scattered base code (Lines 10, 13–15, and 17), alternative return types (Line 9), and parameter variability (Line 14). These cannot be implemented without workarounds in classical modular approaches [8]. Although we could explore the annotative and the modular projection in parallel, it might be beneficial to allow integrating annotation markers into feature modules, as shown in Fig. 3d (statement-level markers in Lines 9–12). To avoid obfuscation, only *Memory_Budget* code is shown. All other code is hidden. This way, the granularity trade-off can be addressed and fine-grained changes implemented. Yet, the surrounding code might be important for comprehension. For instance, the variable `maxByte` in Line 11 of Fig. 3d is declared in the hidden base code from Line 10 of Fig. 3c, which can be shown on demand.

Variant Projection. Now, imagine we want to evolve the features *Base* and *Memory_Budget* or fix a bug that occurs when both features are enabled. We could show all feature artifacts of *Base* by expanding annotation markers in the module *Memory_Budget* (Fig. 3d), or switch to a corresponding variant editor, as shown in Fig. 3e. Among others, this allows exploring variant-specific code and control flows in isolation. To understand which code artifact implements which feature, we can show colored bars (not depicted).

In summary, developers can use the best representation of feature artifacts for an SPL engineering task and observe the impact of changes in real-time—supporting the comprehension of individual features, their combinations, and the whole SPL.

C. PEoPL's Architecture

We implement the PEoPL approach—that is, its internal and external representations—upon the language workbench


a) Textual annotation projection
b) Feature module declaration
c) Visual annotation projection
d) Module projection with annotation marker blending
e) Variant projection (Base and Memory_Budget selected)
Editable Projections (External Representation)
f) Variational AST
rendering
rendering
rendering
Editing Operations Menu
> Assign Variability to ReturnStatement
> Assign Alternative to ReturnStatement
Variational AST (Internal Representation)
vp0 f0 ClassConcept DatabaseImpl
ClassConcept PreloadProcessor
MethodDeclaration preload
MethodDeclaration processLSN
ReturnType f4.1 void
vp4 f4.2 NonOptional Alternative
ReturnType PreloadStats
StatementList
StatementList
vp1 f1 FeatureBlock
AssertStatement assert c ...
vp2 f2 FeatureBlock
IfStatement envImp...
Memory_Budget
vp3 f3 FeatureBlock
IfStatement childType...
Statistics
Feature Block f5 vp5
f6 vp6
f7 vp7
vp8 f8 FeatureBlock
VariableDeclaration cb
ClassCreator
f9 Variable Reference ret vp9
f10 vp10 wrapper TryStatement
Catch Clause
Body
Feature Block f11 vp11
Feature Block f12 vp12
Return Statement ret
Base
...

```
#ifdef Base
class DatabaseImpl {
 static class PreloadProcessor {...}
 #ifdef Statistics
 PreloadStats
 #else
 void
 #endif
 preload(PreloadConfig c) {
   long maxByte = c.getmaxByte();//3) more code
   #ifdef Memory_Budged
   if(maxByte == 0) //4) more code
   #endif
   #ifdef Statistics
   PreloadStats ret = new PreloadStats();
   #endif
   PreloadProcessor cb = new
   PreloadProcessor(envImp, maxByte, tTime
   #ifdef Statistics
   , ret
   #endif
   ); //5) more code
   #ifdef Statistics
   try {
   #endif
     walker.walk();
   #ifdef Statistics
   } catch (HaltPreloadException HPE) {
     ret.status = HPE.getStatus();    }
   #endif //6) more code
   #ifdef Statistics
    return ret;
    #endif
 }}
#endif
```

```
Feature module declaration of Berkeley DB
 Base
 Memory_Budget
 Statistics
```

```
class DatabaseImpl {
 static class PreloadProcessor {
   void processLSN(long c, LogEntTyp t){
     assert c != DbLsn.NULL_LSN
     if(envImp.getMemoryBudget().get())//1) more code
     if(childType.equals(...)) //2) more code
   }
 }
 void ⊕ PreloadStats preload(PreloadConfig c) {
   long maxByte = c.getmaxByte();//3) more code
   if(maxByte == 0) //4) more code
   PreloadStats ret = new PreloadStats();
   PreloadProcessor cb = new PreloadProcessor(
   envImp, maxByte, tTime, ret);
   //5) more code
   try {
     walker.walk();
   } catch (HaltPreloadException HPE) {
       ret.status = HPE.getStatus();
   }
   //6) more code
   return ret;|
 }
}
```

```
module Memory_Budget
refines class DatabaseImpl {
 refines static class PreloadProcessor {
   refines void processLSN(long c, LogEntTyp t){
     original(c, t);
     if(envImp.getMemoryBudget().get())//1) more code
   }
 }
 refines void preload(PreloadConfig c){
   if(maxByte == 0) //4) more code
 }
}
```

```
Variant projection for Base and Memory_Budget
class DatabaseImpl {
 static class PreloadProcessor {
  void processLSN(long c, LogEntTyp t){
     assert c != DbLsn.NULL_LSN
     if(envImp.getMemoryBudget().get())//1) more code
 }
 void preload(PreloadConfig c) {
   long maxByte = c.getmaxByte();//3) more code
   if(maxByte == 0) //4) more code
   PreloadProcessor cb = new PreloadProcessor(
   envImp, maxByte, tTime);
   //5) more code
   walker.walk();
 }}
```

Figure 3. Excerpt of Berkeley DB in PEoPL. Top: projections used by developers (external representation). Bottom: variational AST (internal representation)

JetBrains MPS [39]. MPS relies on a technology called projectional editing (a.k.a., *structured editing* or *syntax-directed editing*), which is conceptually different from parser-based editing [40]. Developers' editing gestures directly change the underlying AST, which is rendered into a concrete syntax. No parsing is involved. While the editing experience is different, and editing efficiency has long been questioned, modern projectional editors allow efficient editing [40], [41].

In MPS, a language's abstract syntax is defined using so-called *language concepts* (a.k.a., *meta-classes*), which define the structure of their instances (AST nodes). Each concept has its own concrete syntax defined by *rendering rules*—a description of how AST nodes appear in a projectional editor.

We realize the languages CoreVar and JavaVar (both are explained shortly) in MPS as follows. The concepts of CoreVar—used to add variability to target languages—are implemented using MPS' core meta-modeling facilities for declaring languages. The JavaVar language—the tailoring of CoreVar to Java—reuses concepts from CoreVar and extends MPS' meta-model of Java. Our five projections are packaged in MPS also as "languages", which contain both target-language-independent and -dependent rendering rules.

IV. PEoPL's Internal Representation

We now present the concepts of PEoPL's internal representation: the CoreVar language, operations for editing the AST, variant derivation, and the tailoring of CoreVar to a target language.

A. The CoreVar Language

Formalism. CoreVar adopts, modifies, and extends the *structured document algebra (SDA)* [42]—an abstract formalization of feature modularity. The SDA enables variability through *feature modules*, which assign *fragments* to *variation points (VPs)*. Let $V = \{vp_1, vp_2, ...\}$ be a set of VPs and $F = \{f_1, f_2, ...\}$ a set of fragments. A module $m : V \rightsquigarrow F$ is an injective partial function assigning fragments to VPs. In contrast to SDA (not using an injective function), fragments are unique to a VP, which is not a limitation, but eases our implementation. The domain $dom(m)$ of a module m is the set of VPs assigned by m. The module m assigns a fragment to the VP vp if $vp \in dom(m)$, otherwise vp is not related to m. A module can assign a VP only once, but multiple modules can assign the same VP. For a fragment $f \in F$, we define the helper function $VP(f)$ returning the VP vp associated with f (possible as of injection). Similarly, let $M(f)$ be the helper function returning the module m assigning f to $VP(f)$. Both helper functions return $\perp$ if there is no module m assigning f to vp (i.e., before a developer explicitly chooses m for assigning f to vp).

Example: For the fragment f_1 in our running example Berkeley DB, $M(f_1)$ yields the feature module *Base* and $VP(f_1)$ the VP vp_1 (cf. Fig. 3f and Fig. 4). All VPs $V_{Berkeley} = \{vp_0, vp_1, ..., vp_{12}\}$ are associated with fragments via modules, for instance $Base : \{vp_0 \mapsto f_0, vp_1 \mapsto f_1, vp_4 \mapsto f_{4.1}, ...\}$. In fact, the feature module *Base* assigns fragments to the VPs in its domain $dom(Base) = \{vp_0, vp_1, vp_4, vp_5, vp_8, vp_{11}\}$. Another example is the feature module $Statistics : \{vp_3 \mapsto f_3, vp_4 \mapsto f_{4.2}, ...\}$ with the domain $dom(Statistics) = \{vp_3, vp_4, vp_7, ...\}$. Note that $dom(Base) \cap dom(Statistics) = \{vp_4\}$ and, thus, *Base* and *Statistics* share vp_4 (cf. Fig. 3f and Fig. 4).

Variational AST. AST nodes are made variable by annotating them with fragments from F. Let $AST = \{n_1, n_2, ...\}$ be a set of AST nodes. A variational AST $vast : F \rightarrowtail AST$ is an injective, non-surjective function assigning AST nodes to fragments from F. The image $vast(F)$ of a variational AST *vast* is the set of AST nodes annotated with fragments. An AST node n is annotated if $n \in vast(F)$. The domain $dom(vast)$ is the set of fragments F (i.e., the *AST* nodes assigned to fragments by *vast*). Due to injection, fragments are unique to AST nodes. Every fragment annotates exactly one node, but not every node must be annotated. The helper function $FN(n)$ either returns the fragment annotating the node $n \in AST$ or $\perp$ if the node is not annotated (i.e., $n \notin vast(F)$).

Example: Fig. 4 shows an excerpt of the mappings in our running example (Fig. 3f). For instance, $vast(F_{Berkeley}) = \{\textsf{FeatureBlock}, \textsf{ReturnType}, \textsf{NonOptionalAlternative}, ...\}$, $FN(\textsf{ReturnType}) = f_{4.1}$, $M(FN(\textsf{ReturnType})) = Base$, and $VP(FN(\textsf{ReturnType})) = vp_4$.

B. Editing Operations

To manipulate the AST, CoreVar provides three basic variability-related editing operations, which can be refined by its tailoring extensions (e.g., JavaVar): *assign variability*, *assign wrapper variability*, *assign alternative*—all available via a menu in the program (cf. Fig. 3c) or triggered automatically by an editing gesture (e.g., typing #IFDEF or #ELIF).

Assign Variability Operation. The operation marks an AST node as variable, such as class `DatabaseImpl` assigned to f_0 (Fig. 3f). An algorithm creates a new VP $vp_i \in V$ and fragment $f_i \in F$, and annotates the selected AST node n_j with the fragment such that $vast(f_i) := n_j$. Then, the developer selects the desired feature module m_k assigning f_i to vp_i.

Assign Wrapper Variability Operation. The operation marks only a wrapping node as variable, not its body (the *wrappee*). The idea is to annotate the wrapping node such that its wrappee

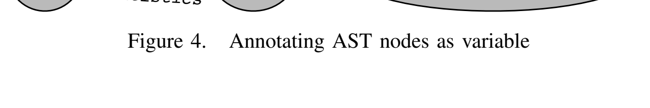


Figure 4. Annotating AST nodes as variable

is not removed during variant derivation (explained shortly). Assigning variability to a wrapping node corresponds to the assign variability operation. The only difference is that an additional annotation called *wrapper* is added to the target node, which refers to the *wrappee* (cf. Fig. 3f, vp_{10}).

Assign Alternative Operation. The operation marks an AST node n_a as alternative to another AST node n_o. Let n_o be variational with $FN(n_o) \neq \perp$, and let its fragment $f_o := FN(n_o)$ be assigned to the VP $vp_o := VP(f_o)$ with $VP(f_o) \neq \perp$. If the alternative node n_a is variational with $FN(n_a) \neq \perp$, then an algorithm changes the alternative node's feature module $M(FN(n_a))$ such that it assigns the fragment $FN(n_a)$ to vp_o, which is then associated with f_o and $FN(n_a)$. If the node n_a is not variational, an algorithm creates a fragment f_a in F, assigns it to vp_o (according to the developer's module selection), and annotates n_a with the fragment such that $vast(f_a) := n_a$. Fig. 4 gives an example. The fragments $FN(\textsf{ReturnType}) = f_{4.1}$ and $FN(\textsf{NonOptionalAlternative}) = f_{4.2}$ are both assigned to vp_4 by their respective feature modules.

Notice that n_o and n_a are typically, but not necessarily siblings in the AST (e.g., statements alternative to each other). In fact, some *non-optional* AST nodes cannot have siblings (e.g., exactly one return type is required in a method declaration according to Java's syntax). For such non-optional nodes, CoreVar provides the concept NonOptionalAlternative, whose instances are used by the assign-alternative operation to annotate n_o, while holding an alternative node n_a. For example, the return type of the method `preload` is annotated by a NonOptionalAlternative holding an alternative return type (cf. $f_{4.2}$ in Fig. 3f). Non-optional language concepts are declared in CoreVar tailorings (explained shortly in Sec. IV-D).

C. Variant Derivation

The derivation of variants is a two-step process. First, we calculate the set of fragments contained in the variant by composing feature modules. Second, we remove any variability by iterating over all fragments in the tree to either remove the fragment, to remove the annotated node, or to restructure the tree (for wrapper and non-optional alternative nodes).

Composition. Composing modules results in a transient *variant set* $F_{variant}$ containing all fragments of a variant. SDA provides three operations to compose modules: *addition* (+), *subtraction* (-), and *overriding* ($\rightarrowtriangle$), which we adopt in PEoPL as follows. Let F_m and F_n be the set of fragments associated with a module m and n, respectively. The *addition* of two modules $m + n$ fails if m and n contain conflicting fragments: $dom(m) \cap dom(n) \neq \emptyset$. In Fig. 4, fragments $f_{4.1}$ and $f_{4.2}$ are conflicting, as both assign fragments to the VP vp_4. Without conflicting fragments, addition results in a greater set of fragments $F_{m+n} = \{F_m \cup F_n\}$ and therefore a larger module serving as input for further operations. Note that the fragments of the larger module reflect a preliminary or the final variant set $F_{variant}$. The *subtraction* of $m - n$ is the set $F_{m-n} = \{f | f \in F_m \wedge VP(f) \notin dom(n)\}$. In other words, we remove the fragments of m that share a VP with fragments of n, so subtraction removes fragments from the variant set.

Table I
EXAMPLE CONFIGURATIONS FOR BERKELEY DB (FIG. 3)

No	Module configuration	Variant's fragment set ($F_{variant}$)	Valid
(1)	*Base*	$f_0, f_1, f_{4.1}, f_5, f_8, f_{11}$	✓
(2)	*Base* + *Memory_Budget*	$f_0, f_1, f_2, f_{4.1}, f_5, f_6, f_8, f_{11}$	✓
(3)	*Base* + *Statistics*	$f_0, f_1, f_3, \lightning, f_5, f_7, f_8, f_9, f_{10}, f_{11}, f_{12}$	✗
(4)	*Base* − *Statistics*	$f_0, f_1, f_3, f_5, f_7, f_8, f_9, f_{10}, f_{11}, f_{12}$	✗
(5)	*Base* ⇾ *Statistics*	$f_0, f_1, f_3, f_{4.1}, f_5, f_7, \ldots$	✓
(6)	*Statistics* ⇾ *Base*	$f_0, f_1, f_3, f_{4.2}, f_5, f_7, \ldots$	✓

Overriding is simply a combination of addition and subtraction: $m \twoheadrightarrow n =_{df} m + (n - m)$ to enable replacement.

Example: Table I shows different compositions for our running example (Fig. 3). For instance, configuration (2), adding *Base* and *Memory_Budget*, results in a valid set of fragments. In contrast, configuration (3), adding *Base* and *Statistics*, is erroneous, as fragments $f_{4.1}$ and $f_{4.2}$ fill the same VP vp_4.

To resolve conflicting fragments, we use subtraction and overriding. For instance, configuration (4) removes $f_{4.1}$. The resulting fragment set is valid, but the AST invalid (both return types pruned), which is detected during derivation. Type-checking module composition is part of our future work. Using overriding, developers decide between conflicting fragments. For instance, configuration (5) denotes that all fragments of *Base* replace those conflicting with *Statistics*. Note that ordering matters for overriding, since configuration (6) includes the fragments of *Statistics* instead. Moreover, notice that a feature selection can be realized by overriding modules according to their declaration order. For instance, configuration (6) reflects a selection of *Base* and *Statistics* based on the ordering defined in Fig. 3b (i.e., *Base* has lowest priority).

Remove Variability. Next, we remove any variability from our *AST* such that $vast(F) = \emptyset$. Fig. 5 shows our algorithm, which takes the variational AST *vast* and the variant's fragment set $F_{variant}$ as input. We iterate over all fragments in the variational AST (Line 2). If the fragment is in $F_{variant}$, we simply delete the fragment to remove variability (Line 3). Otherwise, if the node has a wrapper annotation, we move the wrappee's children up in the tree as siblings of the wrapper (Line 7), and remove the wrapper (Line 15). If the node is non-optional (i.e., has NonOptionalAlternatives), we pop the first alternative and get the node it holds (Line 10), such as the alternative return type `PreloadStats` in Fig. 3f. Since a node may have multiple non-optional alternatives, we must add all of them to the popped alternative (Line 11). Then, we can safely replace the non-optional node in the tree with it (Line 12). If the node neither has a wrapper annotation nor is a non-optional alternative, we remove the node (Line 15).

```
 1 func removeVariability(func vast, set<fragment> F_variant) {
 2   for each f ∈ dom(vast) {
 3     if(f ∈ F_variant) f.delete; // delete only the fragment
 4     else {
 5       if(hasWrapperAnnotation(vast(f)) { // handling the wrapper
 6         for each n ∈ getWrappee(vast(f)).children {
 7           vast(f).add prev-sibling(n); // moving the content up
 8         }
 9       } else if(hasNonOptionalAlternatives(vast(f))) {
10         node a = popFirstAlternative(vast(f)).getTheNodeIHold();
11         a.addAll(getNonOptionalAlternatives(vast(f)));
12         vast(f).replace with(a); // replace the node
13         return;
14       }
15       vast(f).delete; // delete the node and its fragment
16 } } }
```

Figure 5. Algorithm to remove variability from the AST

```
  Can-assign-variability declarations for Java
1 simple inclusion for concepts: Statement, ParameterDeclaration,...
2 parameterized inclusion for node: (sourceNode) -> boolean {
3   return sourceNode.parent.isInstanceOf(BaseMethodDeclaration)
4          && sourceNode.hasRole(BaseMethodDeclaration : throwsItem)
5 }...
6 simple exclusion for concepts: PlaceholderMember, ...
  Can-assign-alternative declarations for Java
7 all rules from can-assign-variability: <default: true>
8 non-optional node concepts: Type, Expression
9 ...
```

Figure 6. Annotatable nodes declaration for Java

D. Target-Language-Specific Tailoring

PEoPL requires tailoring CoreVar to a specific target language. We now explain the tailoring in general and illustrate it with our examples from tailoring to Java.

Annotatable Nodes Declaration. Without restriction, the editing operations of CoreVar allow annotating any AST node (also non-optional ones) with fragments, which may lead to syntactically incorrect variants. To declare "annotatable" nodes and restrict editing operations to a meaningful level, we provide *can-assign-variability* and *can-assign-alternative* declarations. Fig. 6 shows an example for Java. Can-assign-variability declarations are either *simple* or *parameterized inclusions* or *exclusions* of concept instances (and their subconcept instances due to concept inheritance). For example, fragments can annotate Statement concept instances (Line 1) and, thus, all Statement subconcept instances (e.g., IfStatement instances). Moreover, we declare that throwsItems of method declarations can be annotated (Lines 3–4). The can-assign-alternative declaration allows adopting the rules declared in can-assign-variability (Line 7), add new rules, and declare non-optional nodes. For instance, JavaVar allows annotating the language concept instances of Type and Expression with NonOptionalAlternative instances (cf. Fig. 6, Line 8, and the NonOptionalAlternative in Fig. 3f).

Due to concept inheritance, not many declarations are needed for Java. The can-assign-variability declaration has nine simple and eight parameterized inclusions, as well as one simple and three parameterized exclusions. The can-assign-alternative declaration adopts these rules plus two non-optional node inclusions, one parameterized inclusion and four exclusions.

Wrapper Declaration. Which nodes in the AST can have a wrapper annotation—where the wrapping node is variable, but not its subtree (wrapper body)—is target-language-dependent. A wrapper declaration specifies the wrapper's language concept and the corresponding wrappee (child node). Fig. 7 shows the four wrapper declarations for Java. For instance, a TryStatement can be replaced by its body.

```
Wrappers that can be partially annotated in Java
instance of AbstractLoopStatement replaced by its body;
instance of IfStatement replaced by its trueBody;
instance of SychronizedStatement replaced by its block;
instance of TryStatement replaced by its body; ...
```

Figure 7. Wrapper declaration for Java

Further Declarations. JavaVar also declares variability-specific type-system and data-flow rules (cf. [22]). Moreover, it extends the Java language with a convenience concept that eases handling variability. Our FeatureBlock concept groups statements belonging to the same feature module (cf. Fig. 3f). In fact, we enforce that any statement (except partially annotated wrappers) are contained by at least one FeatureBlock (e.g., f_1 in Fig. 3f). Otherwise, all individual statements would need to be annotated (as they are siblings in the AST). In the projections, the block's statement list is just rendered without showing curly braces (cf. statement-level vertical bars in Fig. 3c). A FeatureBlock also extends its enclosing statement list's scope to make the FeatureBlock's statements visible to its siblings. During variant derivation, if the FeatureBlock's module is in the variant, it is replaced by its statements, otherwise removed.

V. PEoPL's External Representations

We use MPS' projection facilities to realize PEoPL's external representations. Each language concept requires a projectional editor defining the rules for rendering concepts into concrete syntax. For instance, a BlockStatement editor renders its StatementList and surrounds it with curly braces. Editors can accommodate so-called *editor hints* defining in which context the rendering rules are to be applied. A language concept can have different editors through different hints. In PEoPL, the variability-related language concept is the fragment. For each of its external representations, we implement a projectional editor that is oblivious to the target language (e.g., Java).

Next, we explain how we realize our projectional editors.

Rendering Annotations. For most target languages projecting textual and visual annotations is easy. Fig. 8 shows a visual annotative editor for our main concept: fragment. Such editor definitions consist of so-called cells [43]. The *[annotated node]* cell embeds the editor of the node annotated with the fragment (Lines 2, 3, and 5). A fragment can be rendered in three ways. First, fragments constituting disciplined annotations are rendered with a vertical bar (Line 2), where *#VerticalBar#* and *#Module#* refer to *editor components*—editors reusable among different editors. Second, undisciplined annotations within a line of code are underlined with a horizontal bar (Lines 3–4). Third, annotations requiring a more specific syntax (e.g., partially annotated wrappers, whose body is not annotated) are propagated to the customized target node's editor (Line 5), which targets a target-language-specific concept (e.g., Java's TryStatement) that recognizes a fragment and provides respective partial coloring. Finally, the textual annotative editor looks similar and just adds keywords, such as #IFDEF.

Rendering Variants. Projecting variants is simple: the editor for fragments checks whether the fragment is in the variant's fragment set. If so, the annotated node is rendered, otherwise

```
1 visualAnnotative editor for concept Fragment
2      if isDisciplined():  #VerticalBar# #Module# [annotated node]
3 else if isWithinLine():   [annotated node] #Module#
4                           #HorizontalBar#
5 else if isComplex():      [annotated node]
```

Figure 8. Simplified visual annotative projectional editor for fragments

hidden (for wrappers the wrappee is shown). To provide a variant-specific file explorer, we simply check for each root node (e.g., class or interface) if it is in the current variant.

Rendering Feature Modules. Modular projections show the code of a feature module in isolation (Fig. 3d). In contrast to projecting annotations and variants, projecting modules is currently language-dependent, since the editor rendering fragments cannot simply hide annotated nodes—that is, we need to show refined structural elements as well (cf. Sec. II). Luckily, it is still feasible to project Java code as feature modules. We only need three simple fragment-aware editors defining how to render the Java language concepts ClassifierMember (e.g., method declarations), IVisible (e.g., *public* or *private*), and StatementList (e.g., a method declaration's or a block statement's body) in the presence of variability. In other words, we override these editors (from Java) with editors that can handle the variability induced by a fragment. We realize these three editors as follows.

First, a fragment-aware ClassifierMember editor renders a classifier member conditionally—if the member or one of its descendants is annotated with a fragment of the module. For instance, the class `PreloadProcessor` (a member of `DatabaseImpl`) is shown in the modular editor of *Memory_Budget*, since a descendant FeatureBlock is annotated with fragment f_2 of *Memory_Budget* (Fig. 3d, Line 3).

Second, to understand whether classes and members are introductions or refinements, we render the keywords *defines* and *refines* into the concrete syntax by overriding the IVisible language concept editor, originally rendering only the member's visibility (e.g., *public* and *private* keywords). For instance, *Memory_Budget* only refines the class `DatabaseImpl` (i.e., in Fig. 3f, f_0 is associated with *Base*) and, thus, the refines keyword is shown (Fig. 3d, Line 2).

Third, we override the editor of the concept StatementList. It filters out FeatureBlock nodes whose fragment does not belong to the current feature module (e.g., the FeatureBlock in Line 6 in Fig. 3c is not shown below Line 6 in Fig. 3d). In case of wrappers not belonging to the module, the wrappee's statement list is the next level to investigate for FeatureBlocks and wrappers. Note that the original keyword is a projection of the base code's FeatureBlock (Fig. 3d, Line 5 and Fig. 3f, f_1). So it is not a real method call as in classical modular approaches and currently restricted to the statement level, which however sufficed in practice. To support the original keyword on the expression level (i.e., as a real method call), we could implement further (but likely more complex) on-the-fly tree transformations and rendering rules.

Finally, note that to provide a module-specific file explorer, we simply check for each file whether it is introduced or refined by the selected module (e.g., a *Memory_Budget* file explorer would show the `DatabaseImpl` file, cf. Fig. 3d).

Combining Renderings. Blended projections simply reuse editor components defined by the projections involved. For instance, to blend annotations into feature modules, elements of the annotative and modular fragment editor as well as horizontal and vertical bars are reused.

Table II
JAVA-BASED PRODUCT LINES ADOPTED IN PEOPL

	Size & Complexity						Scalability & Latency				
SPL	LOC	CLA	MET	F	FM	VP	$\overline{\text{TGV}}$	$\overline{\text{TCV}}$	$\overline{\text{TCF}}$	Source Tool	Description
Jest	19k	144	1105	22	22	205	2535ms	9ms	<1ms	From scratch	Java ElasticSearch client [44]
Berkeley DB	70k	218	3433	42	83	1373	5153ms	45ms	<1ms	CIDE	Embedded database [8]
GPL	1k	15	125	21	26	105	248ms	2ms	<1ms	CIDE	Graph product line [45]
Java-Chat	0,6k	8	58	9	9	33	260ms	<1ms	<1ms	CIDE	Chat client
Lampiro	45k	140	1693	19	19	181	4234ms	7ms	<1ms	CIDE	Instant-messaging client
Prop4J	2k	6	174	14	14	192	249ms	1ms	<1ms	FeatureHouse	Propositional formula libary
Vistex	2k	9	99	16	16	37	287ms	<1ms	<1ms	FeatureHouse	Graph visualization and text editor
STE	1k	9	128	10	10	38	259ms	<1ms	<1ms	DeltaJ	Simple text editor [12], [46]

LOC: lines of code (source) | CLA: classes | MET: method declarations | F : features | FM : feature modules | VP: variation points
$\overline{\text{TGV}}$: time to generate a variant | $\overline{\text{TCV}}$: time to compose a variant | $\overline{\text{TCF}}$: time to compose a file variant

VI. EVALUATION

We now evaluate PEoPL with three objectives.

A. Objectives and Subjects

Objectives. *O1. Analyze expressiveness:* We show that PEoPL can realize SPLs by writing them from scratch or migrating from common annotative or modular variability representations. *O2. Analyze scalability:* We investigate latencies for creating file-variant projections and for deriving full variants, together with qualitatively assessing the editing efficiency. *O3. Assess the benefit of multiple projections:* We study this benefit by analyzing the overhead of a pure modular approach by approximating the boilerplate code it would require to write.

Subjects. Table II shows our SPLs. We migrate seven SPLs used in previous research [8], [12], [16], [45], [47], [48], and implement one (Jest) from scratch. All cover different domains and scales. Most migrations are CIDE projects, for two reasons. First, it is easy to migrate annotative SPLs to PEoPL. We import the codebase and manually re-implement annotations. Second, we aim at using annotative SPLs to evaluate the potential overhead in a pure modular approach. We also migrate three projects from DeltaJ and FeatureHouse, which use modular representations. We import each module as a Java package into PEoPL and use our modular projection for migrating the code. The adoption effort for all subjects is moderate. Creating the subjects takes seven days for Berkeley DB, three days for Jest, and just a few hours each for the others (including comprehending the SPLs).

B. Expressiveness (O1)

Although time-consuming and error-prone, and an analytical approach could have sufficed to evaluate expressiveness, the manual adoption helps us understanding the usability of our projections. No subject requires specific workarounds. We conclude that PEoPL's expressiveness suffices to handle those annotative and modular SPLs. To reduce adoption effort, we plan to write custom importers.

C. Scalability & Latencies (O2)

Metrics. We use the following three metrics (all in milliseconds) to evaluate scalability. TCF (time to compose file) is the time to compose the variant's set of fragments of a single file. TCV (time to compose variant) is the time to compose the variant set of all files (i.e., the complete fragment set). TGV (time to generate variant) includes TCF plus the time to write all Java classes of the variant to disk. We use TGV to compare PEoPL to composition times of other SPL tools.

Methodology. TCV measures the editing latencies of variant projections, since we compose a full product to update the variant editor and explorer (i.e., the tree view on a product's files). PEoPL caches the current variant's fragment set until variability-related operations (e.g., adding a fragment) invalidate the cache. So we turn off caching to avoid confounding. To measure TCV, we compose all feature modules included in the current configuration. TCV is most important, as it excludes the confounding *model-to-text transformation* introduced with TGV. We compare TCV to TCF to determine whether the reduced set of fragments of TCF improves composition performance and yields a better efficiency. A drawback of TCF may be that we need to populate all module-fragment relationships of a file before the composition, since the reduced set is not persisted.

We conduct all measurements on a standard 2011 iMac (3,1GHz Intel i5, 16GB, Radeon HD 6970M, OS X 10.10.5, MPS 3.3.6, Java 1.8) with randomly generated distinct variants. We then compare the composed variant set to all variant sets previously generated. If sets are equal, we skip the current set, otherwise save it for future comparison.

Results. Table II shows all results and Fig. 9 the distribution of TCV values for Berkeley DB (for the others, the values are too low to be meaningful). Generating and writing 2000 Berkeley DB variants to disk is below 5.2 sec. on average ($\overline{\text{TGV}}$). Using equivalent product configurations, we compose and write the same Berkeley DB variant to disk using PEoPL (around 6 sec.), FeatureHouse (around 18 sec.), and CIDE (around 7 sec.). Composing a full variant is below 45 ms on average ($\overline{\text{TCV}}$) and just a single document below 1 ms ($\overline{\text{TCF}}$).

In summary, the PEoPL prototype scales well to SPLs of Berkeley DB size. Latencies to compose the fragments for a variant projection are efficient according to the TCF and TCV measures. PEoPL does not introduce any significant overhead.

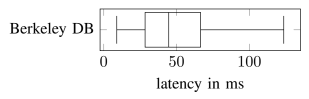


Figure 9. Calculation times for a full variant (TCV)

Table III
METHODS REQUIRING BOILERPLATES IN PURE MODULAR APPROACHES

	Berkeley DB		Lampiro		GPL		Java-Chat	
FM	MET	BOIL	MET	BOIL	MET	BOIL	MET	BOIL
1	2813	0%	1567	0%	77	0%	40	0%
2	398	78%	21	86%	12	25%	6	83%
3	82	86%	8	88%	5	40%	5	60%
4	35	91%	2	100%	3	0%	1	0%
5	21	95%	1	100%	–	–	–	–
6	5	100%	1	100%	1	100%	–	–
7	3	100%	–	–	–	–	–	–
8	1	100%	–	–	2	100%	–	–

FM: feature modules involved in method body | MET: method declarations
BOIL: method declarations that would require boilerplate code

D. Overhead of Pure Modular Representations (O3)

Metrics. The basic idea of O3 is that a developer uses feature modules for their advantages. Yet, classical modular approaches require boilerplate code for fine-grained feature artifacts, such as hook methods. We aim to show the need for annotations, providing indirect evidence that PEoPL—which allows blending annotations into modules and switching to annotations on demand—is useful. To approximate the *potential interest* of these two PEoPL facilities, we measure the number of methods that would require boilerplate code in a pure modular projection in correlation to the involved modules. This shows whether variability in a method's body introduced by different modules impacts the number of boilerplates.

Methodology. To determine the required boilerplates of a method, we search for variability in the middle of a method, and fine-grained variability (e.g., annotated method-call parameters). The method requires boilerplates if such variability is found. Notice that we allow variational method parameters and return-types in our modular projection. Thus, we do not mark such methods as demanding boilerplates (although they would require boilerplates in some classical modular approaches).

Results. It is not surprising that all methods adopted from the modular FeatureHouse and DeltaJ examples do not require additional boilerplates in a pure modular projection. Thus, we concentrate on the CIDE examples. With total numbers of 13% (Berkeley DB), 1% (Lampiro), 6% (GPL), and 15% (Java-Chat), all tested annotative SPLs contain a relatively small number of methods requiring boilerplates. However, as soon as investigating variability in the method's body, the need for blended projections and fluent movement between projections becomes obvious.

Table III shows our boilerplate test results. The majority of methods not requiring boilerplates in pure modular approaches are simple introductions (i.e., only one feature module is involved). The picture changes as soon as a method gets refined (i.e., at least two feature modules are involved). Especially, the Berkeley DB methods require a large number of boilerplates.

E. Threats to Validity

Internal Validity. To mitigate the threat that our SPLs are incorrectly implemented, we cross-checked their implementations and carefully specified and reviewed our generation rules. Moreover, in the final Java code-generation, MPS would have detected invalid ASTs. Furthermore, PEoPL relies heavily on cross-tree references (e.g., fragment to VP). We carefully designed CoreVar to maintain these references throughout the AST editing, as broken references can invalidate a program's variability. Finally, to enhance the validity of our scalability evaluation (O2), we randomly created compositions and implemented a checking rule to detect duplicated ones. We double-checked that tested configurations are not biased (i.e., too few or many feature modules over all configurations). For the modularity overhead (O3), we inspected a sample of nodes to verify that the boilerplates are actually necessary.

External Validity. To increase external validity, seven of our eight subjects are publicly available SPLs of different size and complexity previously used as SPL benchmarks. Among them is Berkeley DB, a substantial embedded database that has been decomposed using different techniques before [8], [16], [48]. Furthermore, although we tested PEoPL only with Java, it is a mainstream language. Java not only benefits from PEoPL's projections, but also from variability support at all. Still, adapting PEoPL to other languages and larger SPLs is valuable future work. Finally, although we evaluated only technical aspects of PEoPL, we can rely on general assumptions in the literature that combinations of multiple representations, views allowing real-time editing, and quick feedback loops, are beneficial. Still, investigating how exactly such views are used when engineering real-world systems is valuable future work, but a study on its own.

VII. LESSONS LEARNED

We made the following experiences that are relevant for using and extending PEoPL in practice.

We found switching representations and using them in parallel useful when adopting our subject SPLs. For instance, we switched the modular to the annotative (or blended) projection when contextual information was required for comprehension. We also switched for implementing fine-grained variability or exploring feature interactions. Using the annotative projection, we found a behavior-related issue in the STE SPL that was neither easy to identify in the original DeltaJ implementation nor in our modular projection [49]. We typically used either the modular and product projection, when annotative code was too complex, or we searched for bugs known to occur in a feature. We also leveraged the locality of the modular projection to identify how a single feature was implemented.

Tailoring PEoPL to a target language using annotatable nodes and wrapper declarations is as easy as *annotating grammars* [16], [50]. The effort for creating the declarations depends on the target language's complexity, but was moderate for Java. Thanks to concept inheritance, languages building upon MPS' Java such as MPS' closure language are inherently supported. Tailoring is also flexible, since even non-textual languages such as MPS' math language with its math symbols are supported.

The realization effort for creating new external representations (rendering rules) is moderate. For instance, it took us only two hours to implement the blended projection. However, implementing, for instance, a modular projection with advanced editing support from scratch requires more engineering effort (e.g., restructuring the tree when typing the original keyword). Likewise, editors that support partially annotated wrappers require customized target-concept editors (for now). So, generating projections and editing rules for variability representations, potentially exploiting the new concept of *grammar cells* [43] for defining projectional editors, would be valuable.

Finally, relying on the concept of projectional editing has the potential of leveraging static analyses that are usually expensive in parser-based systems. Projectional editors operate on an AST (which is directly modified by the user's editing gestures), so references between AST nodes (e.g., method call to method declaration) are actively maintained. This AST can be analyzed for extracting feature constraints. We implemented such a constraint extraction and a preliminary data-flow analysis [22], which already helped to resolve issues (e.g., missing feature dependencies, which would induce deriving invalid variants). In contrast to expensive static analyses required for parser-based systems [51], our analysis is quick (<1,8s on average for Berkeley DB). Using our dependency checker, we in fact found 57 implementation-specific dependencies [22], which were not declared in the feature model of the CIDE version of Berkeley DB. So incorrect variants could have been generated.

VIII. Related Work

Only few approaches exist to integrate annotative and modular variability mechanisms. We discuss such approaches, distinguishing between parser-based and projectional approaches.

The SDA is a formal model of feature modularity [42], [52]. We build upon it, yet use injective partial functions for feature modules to forbid assigning the same fragment to different variation points. To model such (rare) homogeneous extensions [53], [54], we allow variation points to appear multiple times.

The *compositional choice calculus* is a formal language combining annotative and modular techniques [55]. It has been used for a parser-based variant editor [36], which as opposed to PEoPL only has one representation (choice calculus).

Kästner et al. refactor modular (FeatureHouse) into annotative (CIDE) variability representations and vice versa [19]. Yet, refactoring does neither support a fluent movement between techniques nor any other advanced editing support provided by PEoPL. Moreover, parallel editing of different refactored representations can cause inconsistencies, challenging developers.

We previously proposed to integrate variability mechanisms based on variational graphs and outlined the idea of using projections [56]. Using *snippet graphs* [57], [58], we implemented an early version of PEoPL's meta-model to examine the variational graph for Java, Haskell, and HTML SPLs [59]. Now, we present a complete SPL-engineering approach including an IDE with editable projections, evaluated using eight SPLs.

CIDE is a parser-based, annotative approach sharing some concepts with PEoPL, such as annotating ASTs and wrappers [50]. Yet, CIDE only supports an annotative and variant view on the code. New views cannot be plugged and parallel editing is not supported. Moreover, some smaller restrictions exist in the expressiveness (e.g., no mutually exclusive alternative nodes) requiring workarounds that are not required in PEoPL.

DeltaJ is another related modular approach that allows developers to add, replace, and remove feature-related elements by applying deltas [10]–[12]. Instead of incrementally applying deltas, we embed all variants into a single variational AST.

FeatureIDE [60] is an Eclipse framework that integrates several approaches, such as FeatureHouse [16], DeltaJ [12], and tools to cope with variability (e.g., feature-context interfaces [61]). The key difference is that the very same feature artifact cannot be explored using different representations.

Outside the SPL context, *effective views* have been proposed to extract the code from a database into two different text-based views (classes and modules) [62]. Changes made to the concrete syntax can be parsed and merged back into the database. Thus, editing inconsistencies may appear, which is not an issue in projectional editors, since any edit is atomic and directly changes the AST. Moreover, the approach neither supports editing nor generating variants and is tailored/limited to two views not reflecting our notion of features.

Finally, a projectional approach to implement SPLs was proposed before [63], [64]. The language family mbeddr provides the only other projectional way to implement SPLs [65], applying C preprocessor concepts to the underlying AST. mbeddr is also implemented using MPS, but does not focus on providing multiple external representations.

IX. Conclusion

We presented the approach PEoPL, which aims at combining the distinct advantages of different representations of variability. It relies on establishing a unified, internal representation that is separated from multiple external representations. These can be used in parallel and on demand for engineering the same variable software artifact. We designed five complementing representations, allowing developers to edit artifacts using textual and visual annotations, feature modules, a blending of annotations into modules, and to edit individual variants. We realized PEoPL as a full IDE, building upon the projectional language workbench MPS. By declaring annotatable nodes and wrappers, we provide an exemplary tailoring of PEoPL to Java as the target programming language.

We evaluated PEoPL by adopting eight Java SPLs, showing PEoPL's expressiveness, scalability, and benefits. Most importantly, this evaluation shows that it is in fact feasible to separate internal and external representations, supporting very different ways of editing feature artifacts. Latencies to calculate variants are low, which together with our qualitative experiences from adopting the SPLs, evidences a smooth editing experience.

We plan to extend PEoPL with a variability-aware type-checker, version-control facilities for collaborative SPL development, and tailorings to further languages (one to C is on the way). We also plan to conduct user studies investigating exact usage scenarios of multiple projections. Finally, we hope that language designers and tool vendors create further projections, and tailorings for more languages, beyond Java and C.

Acknowledgment

Berger's work is supported by the Swedish Research Council Vetenskapsrådet (257822902) and Vinnova (2016-02804).

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

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

[3] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013.

[4] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines," in *SPLC*, 2015.

[5] D. Le, E. Walkingshaw, and M. Erwig, "#ifdef confirmed harmful: Promoting understandable software variation," in *VL/HCC*, 2011.

[6] H. Spencer and C. Geoff, "#ifdef Considered Harmful, or Portability Experience With C News," in *USENIX*, 1992.

[7] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang, "XVCL: XML-based variant configuration language," in *ICSE*, 2003.

[8] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *ICSE*, 2008.

[9] M. Ribeiro, P. Borba, and C. Kästner, "Feature Maintenance with Emergent Interfaces," in *ICSE*, 2014.

[10] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-Oriented Programming of Software Product Lines," in *SPLC*, 2010.

[11] I. Schaefer and F. Damiani, "Pure delta-oriented programming." in *FOSD*, 2010.

[12] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani, "Deltaj 1.5: Delta-oriented programming for java 1.5," in *PPPJ*, 2014.

[13] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *ECOOP*, 1997.

[14] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[15] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming," in *GPCE*, 2005.

[16] S. Apel, C. Kästner, and C. Lengauer, "Language-Independent and Automated Software Composition: The FeatureHouse Experience," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 63–79, 2013.

[17] J. Melo, C. Brabrand, and A. Wąsowski, "How does the degree of variability affect bug finding?" in *ICSE*, 2016.

[18] J. Favre, "Preprocessors from an Abstract Point of View," in *ICSM*, 1996.

[19] C. Kästner, S. Apel, and M. Kuhlemann, "A model of refactoring physically and virtually separated features," in *GPCE*, 2009.

[20] C. Kästner and S. Apel, "Integrating compositional and annotative approaches for product line engineering," in *McGPLE*, 2008.

[21] J. Siegmund, C. Kästner, L. Jörg, and S. Apel, "Comparing program comprehension of physically and virtually separated concerns," in *FOSD*, 2012.

[22] "Online appendix," http://peopl.de/icse2017.

[23] M. Erwig and E. Walkingshaw, "The Choice Calculus: A Representation for Software Variation," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 1, pp. 6:1–6:27, 2011.

[24] N. Singh, C. Gibbs, and Y. Coady, "C-CLR: a tool for navigating highly configurable system software," in *ACP4IS*, 2007.

[25] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *AOSD*, 2011.

[26] P. C. Clements and C. Kreuger, "Point/Counterpoint: Being Proactive Pays Off - Eliminating the Adoption." *IEEE Software*, vol. 19, no. 4, pp. 28–30, 2002.

[27] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *ICSE*, 2010.

[28] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The Love/Hate Relationship with the C Preprocessor - An Interview Study." in *ECOOP*, 2015.

[29] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.

[30] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[31] ——, "Designing Software for Ease of Extension and Contraction." in *ICSE*, 1978.

[32] C. Kästner, S. Apel, and K. Ostermann, "The road to feature modularity?" in *SPLC*, 2011.

[33] A. D. Baddeley, "Is working memory still working?" *American Psychologist*, vol. 56, no. 11, pp. 851–864, 2001.

[34] B. Kullbach and V. Riediger, "Folding: an approach to enable program understanding of preprocessed languages," in *WCRE*, 2001.

[35] Ş. Stănciulescu, T. Berger, E. Walkingshaw, and A. Wąsowski, "Concepts, Operations, and Feasibility of a Projection-Based Variation Control System," in *ICSME*, 2016.

[36] E. Walkingshaw and K. Ostermann, "Projectional editing of variational software," in *GPCE*, 2014.

[37] B. Westfechtel, B. P. Munch, and R. Conradi, "A layered architecture for uniform version management," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1111–1133, 2001.

[38] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus, "Using version control data to evaluate the impact of software tools: a case study of the Version Editor," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 625–637, 2002.

[39] "JetBrains MPS," https://www.jetbrains.com/mps/.

[40] M. Völter, J. Siegmund, T. Berger, and B. Kolb, "Towards User-Friendly Projectional Editors," in *SLE*, 2014.

[41] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: A controlled experiment," in *FSE*, 2016.

[42] D. Batory, P. Höfner, D. Köppl, B. Möller, and A. Zelend, "Structured Document Algebra in Action," *Software, Services and Systems*, vol. LNCS Volume 8950, pp. 1–21, 2015.

[43] M. Völter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, and T. Berger, "Efficient Development of Consistent Projectional Editors using Grammar Cells," in *SLE*, 2016.

[44] "Jest: Elasticsearch Java Rest Client." https://github.com/searchbox-io/Jest.

[45] R. E. Lopez-Herrejon and D. Batory, "A Standard Problem for Evaluating Product-Line Methodologies," in *GCSE*, 2001.

[46] K. Friesen, "Entwicklung einer Werkzeugunterstützung für DeltaJava und Evaluierung der Sprache anhand einer Fallstudie (German)," Master's thesis, TU Braunschweig, Sep. 2012.

[47] S. Schulze, S. Apel, and C. Kästner, "Code clones in feature-oriented software product lines," in *GPCE*, 2010.

[48] C. Kästner, S. Apel, and D. Batory, "A Case Study Implementing Features Using AspectJ," in *SPLC*, 2007.

[49] B. Behringer and M. Fey, "Implementing Delta-oriented SPLs Using PEoPL: An Example Scenario and Case Study," in *FOSD*, 2016.

[50] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, "Guaranteeing syntactic correctness for all product line variants: A language-independent approach," in *TOOLS EUROPE*, 2009.

[51] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, 2015.

[52] D. Batory, P. Höfner, B. Möller, and A. Zelend, "Features, modularity, and variation points," in *FOSD*, 2013.

[53] A. Rashid, G. Blair, and C. Adrian, "On the separation of concerns in program families," Lancaster University, Tech. Rep. COMP-001-2004, 2004.

[54] S. Apel, T. Leich, and G. Saake, "Aspectual Feature Modules," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 162–180, 2008.

[55] E. Walkingshaw and M. Erwig, "A calculus for modeling and implementing variation," in *GPCE*, 2012.

[56] B. Behringer, "Integrating Approaches for Feature Implementation," in *FSE Doctoral Symposium*, 2014.

[57] L. Kirsch, J. Botev, and S. Rothkugel, "Snippets and Component-Based Authoring Tools for Reusing and Connecting Documents," *Journal of Digital Information Management*, vol. 10, no. 6, pp. 399–409, 2012.

[58] ——, "The Snippet Platform Architecture: Dynamic and Interactive Compound Documents," *International Journal of Future Computer and Communication*, vol. 3, no. 3, pp. 161–167, 2013.

[59] B. Behringer and S. Rothkugel, "Integrating Feature-based Implementation Approaches using a Common Graph-based Representation," in *SAC*, 2016.

[60] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE - An extensible framework for feature-oriented software development." *Science of Computer Programming*, vol. 79, pp. 70–85, 2014.

[61] R. Schröter, N. Siegmund, T. Thüm, and G. Saake, “Feature-context interfaces: Tailored programming interfaces for software product lines,” in *SPLC*, 2014.

[62] D. Janzen and K. De Volder, “Programming with Crosscutting Effective Views,” in *ECOOP*, 2004.

[63] M. Völter and E. Visser, “Product line engineering using domain-specific languages,” in *SPLC*, 2011.

[64] M. Völter, “Implementing feature variability for models and code with projectional language workbenches,” in *FOSD*, 2010.

[65] M. Völter, D. Ratiu, B. Schaetz, and B. Kolb, “mbeddr: an extensible C-based programming language and IDE for embedded systems,” in *SPLASH*, 2012.