A Model of Refactoring Physically and Virtually Separated Features

Christian Kästner
School of Computer Science
University of Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

Sven Apel
Department of Informatics and Math.
University of Passau, Germany
apel@uni-passau.de

Martin Kuhlemann
School of Computer Science
University of Magdeburg, Germany
kuhlemann@iti.cs.uni-magdeburg.de

Abstract

Physical separation with class refinements and method refinements à la AHEAD and virtual separation using annotations à la *#ifdef* or CIDE are two competing implementation approaches for software product lines with complementary advantages. Although both approaches have been mainly discussed in isolation, we strive for an integration to leverage the respective advantages. In this paper, we lay the foundation for such an integration by providing a model that supports both physical and virtual separation and by describing refactorings in both directions. We prove the refactorings complete, so every virtually separated product line can be automatically transformed into a physically separated one (replacing annotations by refinements) and vice versa. To demonstrate the feasibility of our approach, we have implemented the refactorings in our tool CIDE and conducted four case studies.

Categories and Subject Descriptors D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques; D.2.13 [*SOFTWARE ENGINEERING*]: Reusable Software; D.3.4 [*PROGRAMMING LANGUAGES*]: Processors

General Terms Design, Languages, Theory

Keywords Software product lines, refactoring, separation of concerns, preprocessor, refinements, AHEAD, CIDE, FeatureHouse

1. Introduction

A *Software Product Line (SPL)* is a family of related program variants that are generated from a common code base [9, 38, 18]. The generation process facilitates reuse of common software artifacts in different variants and at the same time allows users to tailor each variant to a specific use case. Different variants are distinguished in terms of *features*; a feature represents a user-visible requirement.

There are many different implementation approaches for SPLs. We distinguish [26] between implementation approaches that *physically separate features* (a.k.a. physical separation of concerns) by implementing them in different modules – e.g., plug-ins and components [9, 38] or various flavors of aspects and feature modules [40, 11, 29, 18, 8, 6] – and approaches that *virtually separate features* (a.k.a. virtual separation of concerns) by annotating code fragments in a common code base – e.g., preprocessors [38, 37, 44], XVCL [23], CIDE [26], and commercial SPL tools such as pure::variants [14] or Gears [31]. Virtual separation approaches are often sneered at by academics, because they produce scattered and tangled code instead of pursuing modularity – especially preprocessors are frequently criticized for their undisciplined usage [44, 38, 37, 21]. Nevertheless, virtual separation approaches are common in industry because they are simpler and promise quicker results at lower initial costs [17, 31].

In prior work, we investigated both sides, physical and virtual separation. We addressed typing issues [25, 5], granularity issues [26], or language-independence [6, 28] for both of them. We found that, despite many conceptual differences, both approaches are often similar: for a virtually separated implementation we could often find an equivalent physically separated one and vice versa. In some cases this was straightforward, in others it required more demanding code changes. Still, both approaches have unique advantages, so we cannot simply chose one over the other. For example, a physical separation enables true modularity, but at the price of a more complex implementation and reduced expressiveness compared to a virtual separation (see [24, 26] for a comprehensive discussion). Eventually an integration of both may combine these advantages [24].

In this paper, we lay ground for an integration and a sound comparison of physical and virtual separation by proving that both representations are equivalent to a large degree. We present a formal model that supports both virtual and physical separation at the same time and describe automated refactorings between them. Our goal is to transform a physically separated SPL (with feature modules) into a virtually separated SPL (with annotations) and vice versa without changing the behavior of any program variant. Additionally, the model also supports partial refactorings and mixtures of both representations, so that also SPLs with both annotations and feature modules are possible. This way, developers can use the approach best suited to the task at hand and gradually refactor later [24]. We have exemplarily implemented refactorings between SPL implementations with AHEAD/FeatureHouse-based feature modules [11, 6] and SPL implemented with our preprocessor-based tool CIDE [26] and refactored four case studies. All in all, model and refactorings promise the following benefits:

- We lay ground for an *integration* of virtual and physical separation by providing a model that supports both. This opens new opportunities for *theories, models, and tools* that use both approaches *uniformly*, in contrast to the current practice of searching for solutions for each representation separately.
- Based on this model, we analyze *equivalence* between SPLs implemented in either representation and even provide support to *automatically refactor* one representation into the other.
- Given automatic refactorings, systems decomposed in one paradigm can be used in the other. This lays the ground for

GPCE'09, October 4–5, 2009, Denver, Colorado, USA.

reusing tools developed for only one representation and for future empirical evaluations of both representations regarding understandability, maintainability, development effort or similar aspects on *equivalent* programs.

- The vision behind integration and refactorings is that developers can *leverage respective strengths of both representations*, e.g., start SPL development by annotating code fragments and then gradually refactor them into physically separated modules [24].

2. Background

Before we begin with a description of our formal model and possible refactorings, we provide some necessary background on SPLs and different implementation mechanisms, and we introduce notations for the remainder of this paper.

2.1 Software Product Line Engineering

The aim of SPL engineering is to facilitate reuse in the development of a set of related program variants of a domain [9, 38, 18]. Developers start by analyzing the domain and identify *features*, i.e., requirements that distinguish different variants in the domain [3]. For example, in the domain of embedded database systems, there can be different program variants for different scenarios, but not all variants require features such as transactions, recovery, or ad-hoc query processing. A variant is identified by a selection of features, e.g., the "database system with transactions, but without recovery and query processing". How a variant for a given feature selection is actually generated depends on the implementation technique as discussed below.

Not all *feature combinations* in an SPL are meaningful. For example, features can be mutually exclusive; e.g., users must decide between an *in-memory* or a *persistent* database. Typically, features and their valid combinations are described in a feature model [18, 10].

For this paper, it is not relevant how features and their valid combinations are specified, we assume the following notation and predicates which can be mapped to individual approaches: A feature '*Base*' is part of every SPL and required in every variant. A *feature expression* (denoted with meta-variables F to L) is a propositional formula over features of the product line, e.g., '$Txn \wedge Log$' or '$Inmemory \vee \neg Query$', that evaluates to true or false for a given feature selection. Based on the constraints of the feature model, predicate $mexcl(F, G)$ determines whether two feature expressions are mutually exclusive; predicate $impl(F, G)$ determines whether G is selected in *every* variant in which F is selected; finally, predicate $equiv(F, G)$ determines whether F and G are always selected together in variants (both or neither).

2.2 Virtual Separation of Features

Approaches that virtually separate features in an SPL use a common code base in which code fragments are *annotated* with feature expressions. To generate a variant for a feature selection, some annotated code fragments are removed and the remaining code is compiled. A typical example for virtual separation is the use of the C preprocessor with #ifdef directives as in the small example of a *Stack* with features *Top* and *Undo* in Figure 1 (a).[1]

This example illustrates how virtual separation approaches do not separate code from different variants into different files or modules but scatter them across the entire code base. There are many other tools that pursue similar annotations with different languages or tools like XVCL [23], CIDE [26], or commercial SPL tools like pure::variants [14] and Gears [31]. Virtual separation is common in practice, because it is easy to use, does not require any runtime overhead, and is already natively supported by several programming languages [17]. Nevertheless, especially preprocessor-based implementations are often criticized for their potential complexity, lack of modularity, and reduced readability [44, 38, 37, 21]. Still, some disadvantages like scattered code or potential errors in some variants can be addressed with relatively simple tool support such as discipline constraints [28], views [26], or type checkers [25].

[1] Due to space restrictions, we use a slightly relaxed notation of #ifdef statements throughout the paper. We allow annotations inside a line and propositional formulas like '#ifdef F $\vee$ G' as syntactic sugar for '#if defined(F) || defined(G)'.

```
class Stack {
  #ifdef Undo
  int backup;
  void undo() {/*...*/}
  #endif
  #ifdef Top
  int top() {/*...*/}
  #endif
  void push(int v) {
    #ifdef Undo
    backup=top();
    #endif
    /*...*/
  }
  int pop() {/*...*/}
}
```

(a) virtual separation

Feature Base

```
class Stack { ...
  void push(int v) {/*...*/}
  int pop() {/*...*/}
}
```

Feature Top

```
refines class Stack {
  int top() {/*...*/}
}
```

Feature Undo

```
refines class Stack { ...
  int backup;
  void undo() {/*...*/}
  void push(int v) {
    backup=top();
    original(v);
  }
}
```

(b) physical separation

Figure 1. Minimal Stack with features *Top* and *Backup*

2.3 Physical Separation of Features

The key idea of physical separation is to locate code belonging to a feature or feature expression in a dedicated file, container, or module. A classic example is a framework that can be extended with plug-ins, ideally one plug-in per feature; different variants can be generated by combining different plug-ins [9, 38]. Beyond plug-ins, there is a large body of research on advanced language abstractions to encapsulate features (including crosscutting implementations). Examples are aspects [29], class refinements [11, 6], classboxes [13], and many more. With all these language mechanisms, features can be implemented in separate units (files, containers, modules, ...) and variants are generated from a selection of these units in a composition step.

In this paper, we use a simple language with class refinement-based capabilities close to *AHEAD* [11] and *FeatureHouse* [6], but a mapping to other languages is possible: A class is split into class fragments, and class fragments are located inside *feature modules*. Each feature module is associated with (and identified by) a feature expression. We distinguish between *class introductions* and *class refinements*, the former introduce new classes, while the latter ('refines class ...') can add members to existing classes or extend existing methods. Methods are extended using a *method refinement* mechanism, which can add additional behavior before/after the execution of the original method (denoted by keyword original).

In Figure 1 (b), we show three feature modules implementing the stack example. Class Stack is introduced in the first feature module and subsequently refined by two features (*Top* and *Undo*) to introduce new members. In feature *Undo*, method push is refined to execute an additional statement before the original implementation.

To generate a variant from a feature selection, the feature modules corresponding to the selection are determined and class fragments of these feature modules are merged in a composition step [11, 6]. Note, the order in which feature modules are composed matters, because the order in which method refinements are applied matters. We assume a fixed global order over all feature modules in an SPL (top-down in our listings) and use the predicate $\ll$ to describe the composition order between two feature expressions: $F \ll G$ means that F is composed before G. Furthermore, we assume that feature module *Base* is always composed first. Finally, in line with prior work on composition models [33, 30], we assume that a feature expression $F \wedge G$ is always composed after F and G (i.e., $F \ll (F \wedge G)$ and $G \ll (F \wedge G)$) because it may refine both F and G.

3. Informal Overview

Before we describe our formal model in Section 4, we illustrate the challenges and give an intuition of the desired refactorings between physical and virtual separation by means of examples.

3.1 Physical to Virtual

Both implementations of the Stack SPL in Figure 1 are equivalent, i.e., they obey the same behavior in all variants, as one can easily confirm manually. Our first goal is to refactor one representation into the other. To refactor the physically separated representation (Fig. 1, right) into the virtually separated representation (left), we copy each class introduction with all its refinements flat into a single class. In this process, members are annotated with the feature expression of the feature module they come from; annotations of the *Base* feature can be dropped because the code is included in all variants anyway (see Sec. 2.1). Method refinements (such as push in our example) are inlined such that original is replaced with the refined method body (fresh names if necessary) and the statements of the method refinement are annotated. Multiple refinements of the same method are inlined in feature composition order.

There are more challenging cases when it comes to mutually exclusive features. For example, there might be two implementations of the stack with the same class name as in Figure 2, one on the basis of an array and one on the basis of a linked list, of which exactly one implementation must be selected in every variant.

Feature Array
```
class Stack { ...
  int[] data;
  void push(int v) {/*...*/}
}
```

Feature List
```
class Stack { ...
  Item firstItem;
  void push(int v) {/*---*/}
}
```

Feature Undo
```
refines class Stack { ...
  void push(int v) {
    backup=top();
    original(v);
  }
}
```

(a) physical separation

```
#ifdef Array ∨ List
class Stack { ...
  #ifdef Array
  int[] data;
  #endif
  #ifdef List
  Item firstItem;
  #endif
  void push(int v) {
    #ifdef Undo
    backup=top();
    #endif
    #ifdef Array
    /*...*/
    #endif
    #ifdef List
    /*---*/
    #endif
  }
}
```

(b) virtual separation

Figure 2. Refactoring mutually exclusive features

So, how are two class introductions with the same name merged to get an SPL implementation which includes both features with different annotations? There are different solutions, for example, we could have two class declarations with the same name but different annotations in the virtually separated code. Instead, we prefer to merge class introductions if possible and annotate the resulting class and all shared members with a disjunction of the previous feature expressions (e.g., *Array* $\vee$ *List*). Members located only in one feature module (e.g., data and firstItem) are annotated only with the original feature expression. Members that are introduced multiple times (e.g., push) can be merged similarly. In our approach, we merge classes and members because it avoids code replication when applying further class refinements. In our example, the method refinement in feature module *Undo* is applied only once to the merged method push, instead of applying it separately to both mutually exclusive introductions.

With these simple mechanisms (copy flat, inline, merge), we can refactor class introductions, class refinements, and method refinements all into annotated classes.

3.2 Virtual to Physical

The reverse refactoring from virtual to physical separation is more difficult, since annotations are more expressive, as we will illustrate. The initial steps are simple though: An annotated class is moved to the feature module corresponding to its annotation and merged classes are separated. Classes that are not annotated are moved into feature module *Base*. Each annotated member is moved into a class refinement (created if it does not already exist) in the corresponding feature module. Annotated statements at the beginning or the end of a method are extracted into a method refinement. These steps are the exact reverse of the refactorings above and easy to automate.

However, annotations on statements that do not directly correspond to method refinements are a first challenge. For that reason, reverse operations are not sufficient to refactor all annotated programs. For example, it is possible to annotate statements in the middle of a method, such as sort in the example in Figure 3 (a), which should be executed after inserting data but before the commit call. To separate such code physically, we need to introduce additional explicit extension points (a common strategy in physically separated code [36]) like method hook in Figure 3 (b).

```
class Stack { ...
  void push(int v) {
    a[++size]=v;
    #ifdef Sort
    sort();
    #endif
    commit();
  }
}
```

(a) virtual separation

Feature Base
```
class Stack {
  void push(int v) {
    a[++size]=v;
    hook();
    commit();
  }
  void hook() {}
}
```

Feature Sort
```
refines class Stack {
  void hook() { sort(); }
}
```

(b) physical separation

Figure 3. Refactoring annotated statements inside a method

At this point, we have a choice of what kind of annotations we want to support. Without defining limits, there is no end of possible annotations that must be supported, because every character or token can be annotated. For the small example in Figure 4, even manually, a refactoring into physically separated feature modules is not intuitive to find.

```
class C extends #ifdef A X #else B #endif {
  #ifdef B static #endif int foo(#ifdef C int a #endif) {
    return #ifdef D ( #endif 2 + 2 #ifdef D ) #endif * 2;
  }
}
```

Figure 4. Fine-grained annotations

This shows that there is a trade-off between supported annotations and effort for developing refactorings. We concentrate on those annotations which we found to be common in our projects [26]: annotations on (a) class declarations, (b) members, and (c) statements. We do not consider annotations on the level of expressions, modifiers, or even individual tokens or characters.[2] Although this is a limitation, it allows us to define a concise model which represents the practice and to prove that refactorings are always possible within this model.

A second challenge comes from the fact that the order matters in a physical separation. Composing two feature modules that add method refinements around the same method in different orders can result in different program behaviors. In contrast, in a virtual separation, there is no notion of a composition order; the order is fixed in the common code base. Annotations are evaluated from outer annotations to inner/nested ones. Therefore, it is important to ensure that refactorings into method refinements are performed in a specific order, reverse to the composition order. We can refactor virtually separated code into feature modules in every desired target order, but resulting in different (but behavioral equivalent) implementations [4]. We will come back to this issue later.

4. Formal Model

After having introduced the basic idea behind refactorings between virtual and physical separation, we pursue a formalization. Although the basic mechanisms are simple, the devil is in the details. A one-step refactoring between virtual and physical separation is a complex task, and it is difficult to get all the details and special cases right. Therefore, we use two techniques to make refactorings more manageable:

- We break down refactorings into small steps, small enough to reason about and to give confidence in their correctness. At the same time, we support SPL implementations that use a *combination* of virtual and physical separation. In each small refactoring, we transform only a part of the SPL and thus shift the implementation in the spectrum between virtual and physical separation in one or the other direction. In a final step, we combine the small refactorings (see composite refactorings [42]) and show that they are *complete*, in the sense that we can refactor every program into a pure virtually and a pure physically separated representation.
- We avoid the full complexity of Java (the Java Language Specification is a book with 688 pages of textual specifications), AHEAD and the C preprocessor. Instead, we use a subset of Java based on *Lightweight Java* [45], enriched with basic mechanisms for refinements and annotations. This allows us to focus on the key mechanisms without getting lost in details. In Section 5, we briefly discuss some other Java constructs, not covered in the subset.

Using the above simplifications, we proceed in three steps. First, we introduce a Java subset and extend it with mechanisms for refinements and annotations. Second, we describe small refactorings in both directions step by step and discuss limitations and optimizations. Finally, we show that the refactorings are complete.

4.1 Lightweight Java with Annotations and Refinements

To discuss completeness of our refactorings, we need to define a model of language constructs that we want to support. We use *Lightweight Java*, a subset of Java with classes, fields, methods, and statements, intended "*to be as simple as possible while still retaining the feel of Java*" [45]. In contrast to smaller calculi such as Featherweight Java – which we used in prior work on type-checking both virtually and physically separated SPLs [25, 5] – Lightweight Java contains a larger set of language constructs (specifically statements, including assignments) which makes our refactorings more interesting and a transfer to full Java more realistic. In this paper, we do not repeat Lightweight Java because the internal details of evaluation and typing are not needed (see related work in Sec.7), but its mechanisms become clear from our description.

cd ::=**class** C◁C { $\overline{\text{fd}}$ $\overline{\text{md}}$ }⊣F	*class declaration*
ci ::=**class** C◁C **in** F { $\overline{\text{fd}}$ $\overline{\text{md}}$ }⊣F	*class introduction*
cr ::=**refines class** C **in** F { $\overline{\text{fd}}$ $\overline{\text{md}}$ $\overline{\text{mr}}$ }⊣F	*class refinement*
x,y ::=v | **this**;	*term variable*
fd ::=C f;⊣F	*field declaration*
vd ::=C v;	*variable declaration*
md::=C m($\overline{\text{vd}}$) { $\overline{\text{s}}$ **return** y; }⊣F	*method declaration*
mr ::=**refines** C m($\overline{\text{vd}}$) { $\overline{\text{s}}$ v = **original**($\overline{\text{y}}$); $\overline{\text{t}}$ **return** y; }⊣F	*method refinement*
s,t ::=	*statement:*
{$\overline{\text{s}}$}⊣F	*block*
v = x;⊣F	*variable assignment*
v = x.f;⊣F	*field read*
x.f = y;⊣F	*field write*
if (x==y)⊣F s **else** s'	*conditional branch*
v = x.m($\overline{\text{y}}$);⊣F	*method call*
v = **new** C();⊣F	*object creation*

Figure 5. Syntax of LJ^{AR}

To support annotations and refinements, we make several extensions to Lightweight Java and call the resulting language *Lightweight Java with Annotations and Refinements (*LJ^{AR}*)*. First, we introduce the possibility to annotate classes, members, and statements. Annotations are always defined with respect to their enclosing elements (i.e., a statement is included only if its own annotation and the annotations on parent method and parent class all evaluate to true). An annotation *F* on an element x is written as x⊣F, in which *F* can be a feature expression or empty. An element that is explicitly *not* annotated, is written as x⊣∅. In some refactorings, we omit annotations that are not relevant and propagated unmodified. Second, we introduce constructs for physical separation guided by AHEAD and its formalization in [20]. A class introduction *C* in a feature module for feature expression *F* is written as 'class C extends D in F {. . . }', a class refinement as 'refines class C in F {. . . }' (in a surface syntax, the feature expression is typically specified externally, e.g., represented by a containment hierarchy [11, 6]). A method refinement is similar to a method declaration but has a modifier refines and must contain a single original call. To integrate both physical and virtual separation, also class introductions and class refinements including their members and statements can be annotated. Hence, an SPL can contain both refinements and annotations.

The full syntax of our language is shown in Figure 5. We abbreviate 'extends' as '◁' and use overbars to denote lists, e.g., $\overline{\text{s}}$ is a list of statements, $\overline{\text{x}_i}^{i\in 1..n}$ stands for $\text{x}_1\text{x}_2...\text{x}_n$ (we omit $i \in 1..n$ when the length is not important). We use the meta-variables C, D, and E for class names, the meta-variables *F*, *G*, *H*, *I*, *J*, *K*, and *L* for feature expressions and corresponding feature modules, v for variables, s, t, and u for statements, f for field names, and m and n for method names.

Type safety for product line extensions of calculi like Lightweight Java and Featherweight Java has already been shown [25, 5, 20] and is outside the scope of this paper. In this work, three simple *sanity rules* S.1–S.3 suffice to reasonably discuss correctness. First, we require that two or more classes (class

[2] We believe that it is actually possible to refactor *every* annotated program into a virtually separated one with the same behavior, but depending on the annotation this might require lots of boilerplate code and replication. As last resort, we can always generate a feature module per *variant*, which contains the entire code of this variant, but eliminates reuse entirely. What kind of annotations, beyond those discussed in this paper, can be refactored into more reasonable physically separated representations and whether this has any importance in practice is an open issue.

declarations and/or class introductions) must not have the same name, unless they are defined in mutually exclusive feature modules or have mutually exclusive annotations (S.1). Second, inside a class and its refinements, two or more fields or methods with the same name are not allowed, unless they are defined in mutually exclusive feature modules or have mutually exclusive annotations (S.2). Finally, class refinements must be composed *after* a class introduction with the same name (S.3), see Section 2.3.

4.2 Physical to Virtual

We start with refactorings from physically separated programs or programs that use any combination of physical and virtual separation toward a pure virtual separation. In the resulting program, language constructs from a physical separation (class introductions, class refinements, method refinements) are no longer present.

We start by flattening feature modules into normal class declarations. Because the composition order is relevant in a physical separation, we proceed with one feature module at a time in the composition order. That is, we first refactor all class introductions and refinements of the first feature module into annotated class declarations, then we refactor those from the second feature module, and so on. Step by step, we eliminate class introductions and refinements and create corresponding annotated class declarations.

First, refactoring R.1 takes a class introduction inside a feature module and creates an ordinary class declaration. In this refactoring, annotations have to be changed such that the resulting class and members are included in the same variants as before. In the original code, the class introduction is included in those variants in which feature expression F evaluates to true. Additionally, in case the class declaration has some annotation G, that annotation must evaluate to true as well; in a pure physical separation, there is no such annotation and thus $G = \emptyset$. Thus, the resulting class declaration is annotated with $F \wedge G$ (or just F if $G = \emptyset$). Annotations on members or even statements do not need to be changed, because annotations are defined with respect to their enclosing element, thus changing the annotation on the class implicitly affects all inner annotations.

In case two (or more) mutually exclusive feature modules introduce the same class, with refactoring R.2, we merge all their members into a single class declaration as discussed in Section 3.1. This is the mechanism behind the example in Figure 2. The class declaration is included if either one of the original feature declarations is selected ($F \vee H$). In contrast to R.1, we also need to modify annotations on members, because we include the containing class in more variants than before. Therefore, we propagate the classes' annotations to their members.[3]

Similar to merging classes from mutually exclusive features, with refactoring R.3, we also merge members with the same name inside a class. If a field is introduced in two different feature modules F and G, both instances can be merged and annotated with $F \vee G$. To merge two methods with the same signature, we simply concatenate their statements and propagate annotations as above. Furthermore, since Lightweight Java allows only a single return statement, we need one additional assignment to get the return value right in either case (x and y are return values passed as parameter or assigned in $\overline{\mathsf{s}}$ respectively $\overline{\mathsf{t}}$). Instead of concatenation, we could even implement further optimizations to detect and merge cloned statements.

Next, we transform class refinements with refactoring R.4. Field declarations and method declarations in class refinements are merged like in R.2. We can assume that a class introduction has already been transformed into a class declaration by R.3, otherwise there would be an error in the implementation (violation of sanity

$$\boxed{\textbf{class}\ \mathtt{C \lhd D}\ \textbf{in}\ \mathtt{F\{}\ \overline{\mathtt{fd}}\ \overline{\mathtt{md}}\ \mathtt{\}\dashv G}} \Rightarrow^{\dagger} \boxed{\textbf{class}\ \mathtt{C \lhd D\{}\ \overline{\mathtt{fd}}\ \overline{\mathtt{md}}\ \mathtt{\}\dashv (G \wedge F)}}$$

† provided: class C◁D {...} does not already exist

Refactoring R.1: Move class introduction to class decl.

$$\begin{array}{|l|}\hline \textbf{class}\ \mathtt{C \lhd D}\ \textbf{in}\ \mathtt{F}\ \{ \\ \quad \overline{\mathtt{fd}_i \dashv \mathtt{I}_i}^{i}\ \ \overline{\mathtt{md}_j \dashv \mathtt{J}_j}^{j} \\ \}\dashv \mathtt{G} \\ \textbf{class}\ \mathtt{C \lhd D}\ \{ \\ \quad \overline{\mathtt{fd'}_k \dashv \mathtt{K}_k}^{k}\ \ \overline{\mathtt{md'}_l \dashv \mathtt{L}_l}^{l} \\ \}\dashv \mathtt{H} \\ \hline \end{array} \Rightarrow^{\dagger} \begin{array}{|l|}\hline \textbf{class}\ \mathtt{C \lhd D}\ \{ \\ \quad \overline{\mathtt{fd'}_k \dashv (\mathtt{K}_k \wedge \mathtt{H})}^{k} \\ \quad \overline{\mathtt{fd}_i \dashv (\mathtt{I}_i \wedge \mathtt{F} \wedge \mathtt{G})}^{i} \\ \quad \overline{\mathtt{md'}_l \dashv (\mathtt{L}_l \wedge \mathtt{H})}^{l} \\ \quad \overline{\mathtt{md}_j \dashv (\mathtt{J}_j \wedge \mathtt{F} \wedge \mathtt{G})}^{j} \\ \}\dashv ((\mathtt{F} \wedge \mathtt{G}) \vee \mathtt{H}) \\ \hline \end{array}$$

† provided: $mexcl(F \wedge G, H)$

Refactoring R.2: Merge class introduction with class decl.

$$\begin{array}{|l|}\hline \textbf{class}\ \mathtt{C \lhd D}\ \{ \\ \quad \ldots\ \mathtt{fd} \dashv \mathtt{F}\ \ldots\ \mathtt{fd} \dashv \mathtt{G}\ \ldots \\ \} \\ \hline \end{array} \Rightarrow^{\dagger} \begin{array}{|l|}\hline \textbf{class}\ \mathtt{C \lhd D}\ \{ \\ \quad \ldots\ \mathtt{fd} \dashv (\mathtt{F} \vee \mathtt{G})\ \ldots \\ \} \\ \hline \end{array}$$

$$\begin{array}{|l|}\hline \textbf{class}\ \mathtt{C \lhd D}\ \{\ \ldots \\ \quad \mathtt{E\ m(}\overline{\mathtt{C\ x}}\mathtt{)}\ \{ \\ \qquad \overline{\mathtt{s}_i \dashv \mathtt{H}_i}^{i}\ \ \textbf{return}\ \mathtt{x;} \\ \quad \}\dashv \mathtt{F}\ \ \ldots \\ \quad \mathtt{E\ m(}\overline{\mathtt{C\ x}}\mathtt{)}\ \{ \\ \qquad \overline{\mathtt{t}_j \dashv \mathtt{I}_j}^{j}\ \ \textbf{return}\ \mathtt{y;} \\ \quad \}\dashv \mathtt{G}\ \ldots \\ \} \\ \hline \end{array} \Rightarrow^{\dagger} \begin{array}{|l|}\hline \textbf{class}\ \mathtt{C \lhd D}\ \{\ \ldots \\ \quad \mathtt{E\ m(}\overline{\mathtt{C\ x}}\mathtt{)}\ \{ \\ \qquad \overline{\mathtt{s}_i \dashv (\mathtt{H}_i \wedge \mathtt{F})}^{i} \\ \qquad \overline{\mathtt{t}_j \dashv (\mathtt{I}_j \wedge \mathtt{G})}^{j}\ \ \mathtt{x=y;} \dashv \mathtt{G} \\ \qquad \textbf{return}\ \mathtt{x;} \\ \quad \}\dashv (\mathtt{F} \vee \mathtt{G}) \\ \quad \ldots \\ \} \\ \hline \end{array}$$

† provided: $mexcl(F, G) \wedge x \neq$ this

Refactoring R.3: Merge mutually exclusive members

$$\begin{array}{|l|}\hline \textbf{refines}\ \ \textbf{class}\ \mathtt{C}\ \textbf{in}\ \mathtt{F}\ \{ \\ \quad \overline{\mathtt{fd}_i \dashv \mathtt{I}_i}^{i}\ \ \overline{\mathtt{md}_j \dashv \mathtt{J}_j}^{j}\ \ \overline{\mathtt{mr}_k \dashv \mathtt{K}_k}^{k} \\ \}\dashv \mathtt{G} \\ \textbf{class}\ \mathtt{C \lhd D}\ \{ \\ \quad \overline{\mathtt{fd'}}\ \ \overline{\mathtt{md'}} \\ \}\dashv \mathtt{H} \\ \hline \end{array} \Rightarrow \begin{array}{|l|}\hline \textbf{class}\ \mathtt{C \lhd D}\ \{ \\ \quad \overline{\mathtt{fd'}}\ \ \overline{\mathtt{fd}_i \dashv (\mathtt{I}_i \wedge \mathtt{F} \wedge \mathtt{G})}^{i} \\ \quad apply(\overline{\mathtt{md'}}, \\ \qquad\qquad \overline{\mathtt{mr}_k \dashv (\mathtt{K}_k \wedge \mathtt{F} \wedge \mathtt{G})}^{k}) \\ \quad \overline{\mathtt{md}_j \dashv (\mathtt{J}_j \wedge \mathtt{F} \wedge \mathtt{G})}^{j} \\ \}\dashv \mathtt{H} \\ \hline \end{array}$$

Refactoring R.4: Resolve class refinement

$$\begin{array}{|l|}\hline apply(\\ \quad \mathtt{C\ m(}\overline{\mathtt{C\ x}}\mathtt{)}\ \{ \\ \qquad \overline{\mathtt{s} \dashv \mathtt{H}}\ \ \textbf{return}\ \mathtt{x;} \\ \quad \}\dashv \mathtt{F,} \\ \quad \textbf{wrap}\ \mathtt{C\ m(}\overline{\mathtt{C\ x}}\mathtt{)}\ \{ \\ \qquad \overline{\mathtt{t}_i \dashv \mathtt{I}_i}^{i}\ \ \mathtt{v=}\textbf{original}\mathtt{(}\overline{\mathtt{x}}\mathtt{);} \\ \qquad \overline{\mathtt{u}_j \dashv \mathtt{J}_j}^{j}\ \ \textbf{return}\ \mathtt{y;} \\ \quad \}\dashv \mathtt{G} \\ ) \\ \hline \end{array} \Rightarrow \begin{array}{|l|}\hline \mathtt{C\ m(}\overline{\mathtt{C\ x}}\mathtt{)}\ \{ \\ \quad \overline{\mathtt{t}_i \dashv (\mathtt{I}_i \wedge \mathtt{G})}^{i} \\ \quad \overline{\mathtt{s} \dashv \mathtt{H}}\ \ \mathtt{v=x;} \\ \quad \overline{\mathtt{u}_j \dashv (\mathtt{J}_j \wedge \mathtt{G})}^{j}\ \ \mathtt{v=y} \dashv \mathtt{G} \\ \quad \textbf{return}\ \mathtt{v;} \\ \}\dashv \mathtt{F} \\ \hline \end{array}$$

Refactoring R.5: Apply method refinement

rule S.3 'introduction before refinement', detectable by existing safe composition tools [46]). Nevertheless, special attention is required for method refinements, which change the implementation of existing methods. However, since this mechanism is not trivial, we defer it to an auxiliary function *apply*, which we explain below. After refactoring class refinements, members can be merged again with R.3.

The function *apply* is used to apply a list of method refinements to a list of method declarations and returns a list of (possibly modified) method declarations. Internally, *apply* iterates over all method declarations and checks whether one of the method refinements has a matching signature. If a method refinement matches, it replaces the refined method and the statements of the refined method are inlined at the 'original' call (note, in Java this might require to use fresh variable names); if no method refinement matches, the method is returned unchanged. Due to space restrictions, we show only the core mechanism of applying a method refinement in R.5; the full mechanism of *apply* and how it iterates over multiple method dec-

[3] Note, introducing the same class in two feature modules that may be selected at the same time is an error according to sanity condition S.1; it can be detected prior to our refactoring by existing safe composition tools [46].

```
class C◁D {...}⊣(F∨G)          ⇒†   class C◁D { ... }⊣F
                                     class C◁D { ... }⊣G

class C◁D {                     ⇒†   class C◁D {
  ... md⊣(F∨G) ... }                   ... md⊣F md⊣G ... }
```

† provided: $mexcl(F,G)$

Refactoring R.6: Split merged classes and members

```
class C◁D {fd md}⊣∅      ⇒   class C◁D in Base {fd md}⊣∅
class C◁D {fd md}⊣F      ⇒   class C◁D in F {fd md}⊣∅
class C in F {fd md}⊣G   ⇒   class C in (F∧G) {fd md}⊣∅
```

Refactoring R.7: Move class declaration to feature module

```
class C◁D in F              ⇒†   class C◁D in F
  { ... md⊣G ... }⊣∅                 { ... }⊣∅
```

† provided: $mexcl(F,G)$

Refactoring R.8: Remove dead member

```
class C◁D in F              ⇒†   class C◁D in F
  { ... md⊣G ... }⊣∅                 { ... md⊣∅ ... }⊣∅
```

† provided: $impl(F,G)$

Refactoring R.9: Remove redundant annotations

```
class C◁D in F              ⇒    class C◁D in F { ... }⊣∅
  { ... md⊣G ... }⊣∅             refines class C in (F∧G)
                                   { md⊣∅ }⊣∅
```

Refactoring R.10: Move member to refinement

```
class C◁D in F {                 class C◁D in F { ...
  ...                              D m(D x) {
  D m(D x) {                         t return y;
    s_i⊣G                          }⊣∅
    t                              ...
    u_j⊣G                 ⇒†     }⊣∅
    return y;                    refines class C
  }⊣∅                                     in (F∧G) {
  ...                              wrap D m(D x) {
}⊣∅                                  s_i⊣∅
                                     y=original(x);
                                     u_j⊣∅
                                     return y; }
                                 }⊣∅
```

† provided: $(F \wedge G) \ll H$ with H=feature of last refinement of C.m

Refactoring R.11: Extract method refinement

```
C m(C x) {                       C m(C x) {
  s v=x;⊣F u                       s v=h(v,x); u
  return y;               ⇒        return y;
}⊣G                              }⊣G
                                 C h(D v, E x) {
                                   v=x;⊣F return v;
                                 }⊣G

C m(C x) {                       C m(C x) {
  s v=x.n(y);⊣F u                  s v=h(v,x,y); u
  return y;               ⇒        return y;
}⊣G                              }⊣G
                                 D h(D v, E x, D y) {
                                   v=x.n(y);⊣F return v;
                                 }⊣G
```

Refactoring R.12: Extract statement

larations and method refinements is specified in an accompanying technical report [27].

To summarize, with refactorings R.1–R.5, all class introductions and class and method refinements can be transformed into annotated class declarations. Additionally, we merge mutually exclusive classes and members, which is not necessary (and not always possible; e.g., two classes with the same name but different super types or two methods with the same name but different signatures cannot be merged and are kept as two distinct class declarations with different annotations) but useful to avoid replication as discussed in Section 3.1.

4.3 Virtual to Physical

Next, we discuss refactorings in the other direction, in which we replace all annotations (from a pure virtual separation or from a program that uses both annotations and feature modules) with class introductions and class refinements. Unfortunately, this is not 'just' the reverse operation, because many annotations are possible (even in our limited model) that have no immediate representation using feature modules [26]. We start by separating members into different class introductions and refinements, and afterward discuss how to handle annotated statements inside a method. Except for refactoring R.11, the order in which these refactorings are applied does not matter, as we will discuss.

As a first step, refactoring R.6 splits classes and members annotated with a disjunction of mutually exclusive feature expressions. This is inverse to R.2 and R.3, except that we defer splitting statements inside methods to refactorings R.11 and R.12 below.

Second, refactoring R.7 transforms each class declaration directly into a class introduction. After this refactoring, every class is still included in the same variants, but annotations are no longer required. If a class declaration does not have any annotation, it is moved into the *Base* module and thus included in all variants.

Refactorings R.8–R.10 eliminate annotations on methods (annotations on fields are removed exactly the same way and omitted here for brevity). There are three possible cases:

1. A method is dead. A dead method is annotated in such a way that it is *never* included in any variant that includes the class (potentially caused by merging or by user-defined annotations). Dead members are removed with R.8.
2. A method is *always* included in the same variants as the enclosing class. In this case, the annotation is redundant and can be removed with R.9 (necessary to revert R.2).
3. A method is included in *some* variants. These methods are moved into class refinements with R.10 (reverse of R.4, except we do not split methods yet). Class refinements are created in this refactoring as needed; if a suitable class refinement already exists, the member is moved there. Note, the composition order of the resulting feature models automatically fulfills sanity rule S.3 ('refinement after introduction'), because $F \ll (F \wedge G)$ (see Sec. 2.1).

So far, refactorings R.6–R.10 split classes and members and move them all into their respective feature modules. They can be executed until all annotations on classes and members are eliminated. The part that is technically the most difficult is eliminating annotations on statements by extracting one or more method refinements (dead statements and redundant annotations can be resolved as above for methods). The reverse operation of applying a method refinement (R.5) is simple, but only works if the first and/or last statements belong to a feature like in Figure 1. This would allow us to refactor all programs that originate from a physically separated program, but unfortunately (or fortunately, depending on the point of view) virtual separation is more flexible and expressive because there is no feature order and because, like in Figure 3, statements in the middle of a method can be annotated [26, 24]. That is, in

```
class Foo {
  C m(...) {
    ...
    #ifdef A
    y=z;
    x=z;
    #endif
    ...
    return x;
  }
}
```

```
class Pxy { C x; C y; }
class Foo {
  C m(...) { ...
    p=h(p, x, y, z);
    x=p.x; y=p.y;
    ... return x;
  }
  Pxy h(Pxy p, C x, C y, C z) {
    #ifdef A y=z; x=z; #endif
    p=new Pxy(); p.x=x; p.y=y;
    return p;
  }
}
```

Figure 6. Extracting sequences of statements

many cases, we cannot directly extract method refinements, but we need a step to prepare the source code. In the following, we describe general steps for refactoring annotated statements. We keep the steps described here intentionally simple and aim primarily at completeness. There are many possible optimizations to create a less verbose output, some of which we discuss below.

First, we extract method refinements as long as possible with refactoring R.11 (works in class introductions and class refinements [27]). Technically, there are three conditions to extracting method refinements: (1) the first and/or last statements must be annotated, (2) annotated statements at the end of the method must not access variables modified by the inner statements (except the return value), and (3) if we already extracted a method refinement from this method, the target feature module must be composed before the feature module that contains the previously extracted method refinement. The second condition is required because assignments to variables in inner statements are not visible to a method refinement. The third condition is important to retain the order in which statements are executed. The outermost method refinement must be extracted first and applied last; if this order cannot be guaranteed, we need a different solution.

A solution for all cases in which it is not possible to apply R.11, is to extract annotated statements into a separate method *h* (any fresh name) each. This is essentially an instance of the well-known extract method refactoring [22]. In the extracted method, the statement is the *only* statement and can be extracted as method refinement with R.11 (i.e., all three conditions are fulfilled). We formalize this mechanism in R.12 for assignments and method calls and skip similar refactorings for the other statements due to space restrictions (see accompanying technical report [27] for a full list).

Instead of extracting every statement in isolation, it is also possible to extract sequences of statements (also necessary for blocks and conditionals). The challenging part for sequences is to get the parameters and return values right, which requires static source code analysis (e.g., define-use chains). As parameters, we need every variable that is ever accessed (read or write) inside the block/conditional. Furthermore, we need to return every variable that is written and accessed later. Because Java can have only a single return value, we need to construct (generate) complex return objects that are assigned to individual variables later. For an example that can be extrapolated into a general pattern, see Figure 6.

To summarize, we use refactorings R.6–R.10 to eliminate annotations on classes and members and refactoring R.11 to eliminate annotations on statements. If R.11 cannot be applied directly, we can extract the annotated statement into a dedicated method first.

Optimizations: Finally, some optimizations are possible, to make the refactored code less verbose. First, if a sequence of statements is annotated with the same feature expression, it is possible to extract the whole sequence instead of every item in isolation to reduce the number of generated methods. Second, it is a good idea to mark the generated boilerplate code (either with a naming convention or another mechanism like Java's annotations) and to remove or inline this code when refactoring the generated code back into a virtually separated representation. Finally, methods and method refinements are moved into feature module $F \wedge G$ with R.10 and R.11. In physically separated programs, complex feature annotations are less common. Nevertheless, the conjunction $F \wedge G$ is necessary to ensure that refinements are composed in the correct order ($F \ll F \wedge G$, see Sec. 2.1). If G is composed after F ($F \ll G$) and G is always selected when F is selected ($implies(G, F)$), we can move the method/method refinement into feature module G instead of $F \wedge G$.

4.4 Completeness

After formalizing the individual refactorings, the question arises whether, in combination, the refactorings are complete. That is, we want to show that every possible program (given the syntax and sanity rules of LJ^{AR}) that uses any combination of feature modules and/or annotations can be refactored into a pure virtual separation (with annotations, without feature modules) and also into a pure physical separation (with feature modules, without annotations).

For our refactorings, completeness is actually straightforward to show. Let us start again with refactorings toward a pure virtual separation. With R.1 and R.2, we can eliminate all class introductions, leaving only class refinements and annotated class declarations. The only possibility this could fail is if two classes with the same name were not mutually exclusive, which would be a violation of sanity rule S.1 in the first place. Refactoring R.3 reduces replication, but is not necessary to show completeness. Finally, refactoring R.4 finishes the case, because it eliminates all class refinements and applies all method refinements. We can ensure that R.4 eliminates *all* refinements, because sanity rule S.3 specifies that the class refinement must follow a class introduction with the same name (which would be refactored by R.1 or R.2 into a class declaration first).

To show completeness for refactorings toward a pure physical separation, we begin with statements. By applying R.12, we can extract every annotated statement into a separate method. Refactoring R.12 works without additional conditions for all possible statements in LJ^{AR}. After they have been extracted into separate methods, the original method does not contain any annotated statements, and the new methods contain only a single annotated statement each, which can be subsequently extracted into a method refinement by refactoring R.11. This way, R.11 and R.12 eliminate all annotations on statements, leaving us with annotated classes and members that – without further conditions – can be refactored into class introductions and class refinements with R.7 and R.10, thus removing the remaining annotations and finishing the case. □

5. Implementation & Case Studies

We have implemented the refactorings between virtual and physical separation as *exports* and *imports* in our tool CIDE [26]. CIDE is a preprocessor-like environment for SPLs based on Eclipse, in which code fragments can be annotated and different variants can be generated from this annotated code. In contrast to a traditional preprocessor as in C, CIDE enforces 'disciplined' annotations, i.e., only entire classes, methods, or statements can be annotated (similar to annotations in LJ^{AR}, see Fig. 5). Annotations are stored by a tool infrastructure and are visualized with background colors in the editor.

The refactoring from virtual to a physical separation is implemented as export in CIDE. Currently, CIDE supports to export annotated Java code into AHEAD [11], FeatureHouse [6], and AspectJ [29] feature modules. Internally, this export is performed by AST transformations based on the annotations as described in our model above. The mechanism is similar for all target languages, differences lie mostly in the surface syntax of the result, i.e., how class and method refinements are specified.

The refactoring from physical to virtual separation is implemented as import in CIDE. CIDE can import AHEAD and FeatureHouse modules and refactor them into annotations. Since these languages use refinement mechanisms very close to LJ^{AR}, we took the existing implementation of the FeatureHouse composition engine and extended it to support the composition of mutually exclusive feature modules and to propagate associated feature expressions during the composition.

In addition to the refactorings in this paper, we implemented several additional refactorings for language features that are frequently needed and annotated in our SPLs, but that are not part of Lightweight Java (and LJ^{AR}). Specifically, we added support for local variables (that are tricky to refactor, but possible with some static source code analysis and boilerplate code) and for no or multiple return and original statements that must not necessarily be top-level statements as in LJ^{AR}. A thorough discussion of these additional refactorings is outside the scope of this paper.

Since its development, we have used our refactorings for a couple of practical applications. For instance, for some recent work on a language extension of AHEAD [32], we wanted to decompose a number of legacy applications from different domains into SPLs. We used CIDE to annotate and subsequently export AHEAD code, because we felt that this would be much faster than extracting class refinements manually. Also the Berkeley DB case study in [6] was created this way. We also imported existing projects, to use CIDE's type-checking mechanism [25] on existing, physically separated SPLs.

In the following, we report some statistics of four projects we refactored. To avoid biased decompositions, we describe only refactorings of SPLs that have been developed prior to our implementation (and, except Berkeley DB, by other authors). Due to space restrictions, we limit our discussions on brief statistics shown in Table 1:

- First, we imported (physical to virtual) the common SPL example 'graph product line', proposed in [34] as a benchmark for SPL technology. We imported an implementation with 2000 lines of AHEAD code and 20 features in 29 physically separated feature modules (4 pairs of mutually exclusive features, 6 optional features) and exported it back again.
- Second, we imported the *Bali product line* which is a set of AHEAD tools to manipulate, transform and compose grammars. Bali was implemented with 18 physically separated feature modules, with about 7000 lines of AHEAD code [11] (see [46] for feature model). In Bali there are three mutually exclusive and several optional features to generate different tools.
- Third, we exported Berkeley DB, an embedded database engine with 80 000 lines of Java code, which we virtually separated with CIDE into 38 features in earlier work [26]. This way, we refactored the annotated code into feature modules and back again. Due to many annotations in the form $A \wedge B$ this resulted in 99 exported feature modules.
- Finally, we exported an annotated version of Prevayler (an object persistence library) with 5 features and 8000 lines of Java code and imported it back again. Prevayler was annotated by V. B. de Oliveira independently of our work.

In Berkeley DB and Prevayler, there were a few annotations not supported by our refactorings (especially some annotated parameters), so we prepared the code slightly (using overloaded methods instead of annotated parameters).

We refactored all SPLs in both directions. Exporting an SPL and importing it back again does not necessarily yield exactly the same program. Aside from whitespace differences, some refactorings create boilerplate code (e.g., additional assignments in R.3 and R.4, additional methods or classes when extracting statements). Some of this boilerplate code is removed in the reverse refactoring, but some remains because it is not always straightforward to decide whether code is user-written or generated. Nevertheless, in all SPLs, we sampled a number of variants from the original and the refactored SPL implementations. Although the variants are not necessarily syntactically equivalent, we used runtime tests to confirm that they behave equivalently. CIDE is available online: `http://fosd.de/cide`.

		Virtual Sep.			Physical Sep.		
SPL	FE	CD/MD	AN	D	FM	CR/MR	GH
GraphPL	20	16/163	167	←	29	41/29	-
Bali	18	40/503	122	←	18	26/9	-
Berkeley	38	283/6515	2297	→	99	338/954	858
Prevayler	5	140/994	175	→	8	13/19	28

FE: number of features; CD/MD: class/member declarations; AN: annotated code fragments; D: direction of initial refactoring; FM: feature modules; CR/MR: class refinements/method refinements; GM: generated 'hook' methods

Table 1. Statistics before and after refactoring

6. Discussion & Perspective

An insight, confirmed by this formalization, is that annotations are more expressive than a physical separation: Annotations are able to implement more fine grained extensions, e.g., statements in the middle of a method, parameters, or even arbitrary tokens [26]. In contrast, most approaches for a physical separation provide coarse-grained mechanisms like method refinements. A refactoring that can transform any possible annotation, such that any sequence of characters can be annotated, appears not worth pursuing. Even if we found such a refactoring, the effort for its implementation and the complexity of the generated code (that has to be implemented with coarse grained mechanisms like method refinements by using workarounds like preliminary refactorings) would render such approach infeasible. However, as we have shown, we can define refactorings and prove them complete if we limit the expressiveness of annotations to 'disciplined' annotations.

Formally, 'disciplined' annotations reduce the expressiveness of a virtual separation. Nevertheless, this approach has often even been discussed as beneficial regarding readability. According to studies by Ernst et al. [21], Baxter and Mehlich [12], and Vittek [47] in practice most annotations are already in a disciplined form (66–85 %), and developers typically strive for disciplined annotations (*"The reaction of most staff to this kind of trick is first, horror, and then second, to insist on removing the trick from the source."* [12]). Unless there is a policy that forbids to change legacy code, disciplined annotations are not significantly limiting: According to Baxter and Mehlich refactoring annotated legacy annotations into disciplined annotations for 50K LOC of C code can be done within few hours [12].

Nevertheless, the question remains: Which kind of annotations and which kind of language constructs from physical separation should be supported? For example, should we allow annotating parameters? Or should we consider quantification mechanisms from contemporary aspect-oriented languages [35]? As usual there is a balance between complexity, readability, and effort for implementing refactorings. Especially evaluation regarding source code complexity and readability requires empirical studies, which are still missing [7]. In our work, we decided to support a sound set of language constructs, guided by (a) capabilities of AHEAD, FeatureHouse, and similar tools and (b) by our experience with frequently used constructs from earlier projects [26, 2].

With LJ^{AR}, we have demonstrated that (within the limitations of 'disciplined' annotations) both virtual and physical separation can express the same programs. This allows us to leverage previous comparisons that pointed out respective advantages of both approaches and use a combination of both. For example, regarding SPL adoption, annotations are considered to be quicker and less risky, but physical separation is considered to be better suited for long term development and maintenance [17, 24]. By supporting both representations and being able to refactor between them, we can start with a virtual separation and gradually refactor toward an physical separation, thus combining both advantages and lowering the adoption barrier.

Another point worth mentioning is that some refactorings require workarounds or boilerplate code (e.g., complex feature expressions or generated statements, methods, classes), which may have a negative impact on readability. There will always be implementations that are more readable in the one or the other representation. Again, the benefit of automated refactorings is that we can have both representations and the developer can decide which one to use for each task.

Finally, there are numerous tools and theories that have been developed for one or the other representation, e.g., navigation tools and views on annotated source code [43, 26] or approaches to analyze feature interactions in feature modules [33, 46]. With an integration and automated refactorings, we can reuse them for either representation.

7. Related Work

There are five fields of related work: (1) extracting features from from legacy code, (2) refactoring preprocessor code into physically separated code, (3) refactoring from physical to virtual separation, (4) composition order, and (5) type-checking SPLs.

First, there is a group of approaches that begin with legacy code and turn it into an SPL by identifying and extracting features. The key difficulty lies in locating the code that belongs to a feature, a process known as feature location or aspect mining [39, 15], and not in the actual refactoring. Once, feature code has been identified, there are additional questions regarding interacting or overlapping features, i.e., code fragments belonging to multiple features. For such situations, models for multidimensional feature structures have been developed, most prominently lifters [40] and derivatives [33, 30], which all create additional feature modules that belong to complex feature expressions (e.g., $F \wedge G$). Our work builds on these results and underlying composition models (many of our refactorings create code fragments annotated with a conjunction of features), but focuses on automated refactoring of already separated code, not on locating and extracting new features.

Second, there are several related approaches to (partially) refactor virtually separated legacy code automatically into a physical separation. Especially in the field of aspect-oriented software development, there has been effort in transforming *#ifdef* statements in legacy C programs into aspects [1, 16, 41]. The key concern is to understand existing preprocessor usage, e.g., classify what typical patterns exist and how they can be extracted [1, 16, 41]. Many approaches eventually enforce disciplined annotations [12, 41] or parse code only partially, while ignoring undisciplined annotations [1]. Furthermore, these approaches usually do not consider alternative features. In contrast, our work does not aim at understanding all legacy code, but we consider only SPLs with disciplined annotations. Nevertheless, this enables us to *guarantee* that every possible disciplined annotation can be refactored.

Third, refactorings from physical to virtual separation are rare, because most researchers regard a physical separation as the more desirable form. The only exception we are aware of is the work of Kim et al. [30], who discuss differences regarding ordering and type-checking for virtual and physical separation. In their work, they mention that they have mechanically transformed AHEAD projects into an annotated code base to create their case studies, but this transformation is not formalized and alternatives were not discussed.

Fourth, there have been discussions about the importance of the composition order in physically separated programs and whether the same program can be rewritten to use a different composition order [4, 30]. With the notion of pseudo-commutativity there are transformations to switch the order of two features by changing their implementation but not their behavior (e.g., by introducing hook methods as in R.12). Interestingly, our refactorings corroborate this theory and can actually be used to perform pseudo-commutative transformations: We can refactor a physically separated program in one order into a virtually separated one (which does not have a notion of order) and back to a physically separated program in any desired order.

Finally, there are several approaches to type check SPLs, i.e., find typing errors as dangling method invocations in all variants without actually generating them all. There are calculi for both virtual [19, 25, 30] and physical separation [46, 5, 20]. A challenge for future work is to model a calculus that supports both representations and formally prove that our refactorings preserves semantics and typing. In this work, we limited our discussion to few essential sanity conditions (S.1–3) from these calculi and gain confidence in the correctness of our refactorings from splitting them into small steps, as it is common for refactorings [22, 42].

8. Conclusion

We have presented a formal model for a programming language LJ^{AR}, which supports both virtual separation of features, using annotations (à la *#ifdef* or CIDE) and physical separation of features, using feature modules with refinements and method refinements (à la AHEAD or FeatureHouse). Based on this model, we have described refactorings to transform any given SPL that uses either representation or even a mixture of both toward a pure virtual or a pure physical separation. We have implemented these refactorings in CIDE and demonstrated practicality on four case studies.

In the context of our model, we have shown the equivalence between both representations and proved the refactorings complete. This lays ground for an integration of different SPL development methods and tools allowing developers to select the representation suited best for the problem at hand, while still allowing to change the representation later. In future work, we intend to build a tool infrastructure that, like LJ^{AR}, supports virtual *and* physical separation and small step refactorings between them. Additionally, we plan to empirically evaluate the benefits of either representation on program comprehension; our refactorings provide a foundation for this.

Acknowledgments. We thank Don Batory for discussions about different refactorings and implementation patterns, including many tricky details regarding alternative features, which eventually motivated us to pursue a formalization. Furthermore, we thank Virgilio Borges de Oliveira for providing an annotated version of Prevayler and the anonymous GPCE reviewers for providing valuable feedback. Apel's work is supported in part by DFG project #AP 206/2-1.

References

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

[2] S. Apel. How AspectJ is Used: An Analysis of Eleven AspectJ Programs. *Journal of Object Technology (JOT)*, 9(1), 2010.

[3] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[4] S. Apel, C. Kästner, and D. Batory. Program Refactoring using Functional Aspects. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 161–170. 2008.

[5] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 101–112. 2008.

[6] 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.

[7] S. Apel, C. Kästner, and S. Trujillo. On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns. In *Proc. ICSE Workshop on Assessment of Contemporary Modularization Techniques (ACoM)*, 2007.

[8] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng. (TSE)*, 34(2):162–180, 2008.

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

[10] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 7–20. 2005.

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

[12] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290. 2001.

[13] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 177–189. 2005.

[14] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Sci. Comput. Program.*, 53(3):333–352, 2004.

[15] S. Breu. Aspect Mining Using Event Traces. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 310–315. 2004.

[16] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple Crosscutting Concerns Are Not So Simple: Analysing Variability in Large-Scale Idioms-Based Implementations. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 199–211. 2007.

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

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

[19] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. 2006.

[20] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, 2009.

[21] M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.

[22] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[23] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 810–811. 2003.

[24] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40, 2008.

[25] C. Kästner and S. Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 258–267. 2008.

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

[27] C. Kästner, S. Apel, and M. Kuhlemann. LJ^{AR}: A Model of Refactoring Physically and Virtually Separated Features. Technical Report 8/09, School of Comp. Science, University of Magdeburg, Germany, 2009.

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

[29] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242. 1997.

[30] C. H. P. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 23–34. 2008.

[31] C. Krueger. Easing the Transition to Software Mass Customization. In *Proc. Int'l Workshop on Software Product-Family Eng.*, pages 282–293. 2002.

[32] M. Kuhlemann, D. Batory, and S. Apel. Refactoring Feature Modules. In *Proc. Int'l Conference on Software Reuse*. 2009.

[33] 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.

[34] R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l Conf. Generative and Component-Based Software Engineering*, pages 10–24. 2001.

[35] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 2–28. 2003.

[36] G. Murphy et al. Separating Features in Source Code: an Exploratory Study. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 275–284. 2001.

[37] D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Proc. Net.ObjectDays*, pages 313–329. 2003.

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

[39] D. Poshyvanyk et al. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Trans. Softw. Eng. (TSE)*, 33(6):420–432, 2007.

[40] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443. 1997.

[41] A. Reynolds, M. E. Fiuczynski, and R. Grimm. On the Feasibility of an AOSD Approach to Linux Kernel Extensions. In *Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 1–7. 2008.

[42] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[43] M. Robillard and G. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 406–416. 2002.

[44] H. Spencer and G. Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, pages 185–198, 1992.

[45] R. Strniša, P. Sewell, and M. Parkinson. The Java Module System: Core Design and Semantic Definition. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 499–514. 2007.

[46] 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.

[47] M. Vittek. Refactoring Browser with Preprocessor. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 101–110. 2003.