Integrating Feature-based Implementation Approaches using a Common Graph-based Representation

Benjamin Behringer
University of Luxembourg, Luxembourg
htw saar, Germany
benjamin.behringer@htwsaar.de

Steffen Rothkugel
University of Luxembourg, Luxembourg
steffen.rothkugel@uni.lu

ABSTRACT

Based on the structured document algebra, we propose the model of structured document graphs: a common, generic graph-based representation for compositional and annotative feature implementations. The core elements in the graph are modules that allow adding, removing and replacing feature details in a fine-granular fashion. To show the feasibility and generality of our model, we prototypically implemented our concepts and imported five FeatureHouse examples (written in Java) and three CIDE projects (written in Java, Haskell and HTML) into our prototype. To test the validity of the resulting graph, we randomly produced different products, semi-automatically compared our results to the ones produced by the original tool, and found no semantic differences. As a result, our model lays the foundation for projections that allow moving fluidly between compositional, annotative and mixed versions of the product line.

CCS Concepts

•**Software and its engineering** → *Abstraction, modeling and modularity; Software product lines;*

Keywords

Feature-oriented software product lines, compositional approach, annotations, structured documents, variability

1. INTRODUCTION

A *feature-oriented software product line* is a set of software systems in which end-user visible features define common and variable artifacts [1]. From a feature selection, a product variant can be generated automatically. To make features explicit in the product line's implementation, there are two different ways: *compositional* and *annotative* ones [10].

Compositional approaches separate feature implementations physically into modules (e.g., in AHEAD [4] and FeatureHouse [2], a directory represents a module). To produce a product, modules can be composed. *Annotative approaches* like the C-preprocessor (CPP), C-CLR [23] and CIDE [11] separate feature implementations virtually by annotating, for instance, source code fragments. Annotations are either *textual markers* in the concrete syntax (e.g., `#ifdef`) or *visual markers* that are externally stored in a tool infrastructure (e.g., *coloring* [11]). To produce a product, annotated fragments can be excluded conditionally.

Both implementation approaches share well-known distinct advantages [10–13, 19, 26]. For instance, compositional approaches inherently provide locality and cohesion, while annotative ones allow fine-grained feature extensions without requiring workarounds. While current research strives for an integration of the two [3, 10, 12, 26, 27], there seems to be no full-fledged applied integration available yet that

- supports arbitrary structured documents such as source code, HTML pages, makefiles and diagrams;
- allows moving fluidly between compositional, annotative and mixed versions of the product line.

In this paper, we provide a first step to address this lack. We propose a logical model (i.e., abstract syntax) for a promising full-fledged, but so far theoretical approach: the *structured document algebra* (SDA), a mathematical formalization of feature modularization that covers any kind of *structured document* [3]. SDA acts in solution space and formalizes *variation points* (VPs), *fragments* and *modules*, and the composition of these modules. Similar to *join points* in AOP, a VP represents a potential point of variability in a product line. A module is a partial function that assigns fragments/content to VPs. SDA allows that the same VP appears multiple times across a program. Then, a fragment is a single extension applied to multiple join points.

Our variant of SDA is a logical model, which enables the representation of structured documents as a so-called *structured document graph* (SDG). This graph is built from interrelated VPs, fragments, modules, and—in contrast to SDA—structural elements. We think that this variation of SDA is more practical for reflecting structured content. In particular, we decompose a fragment into a smaller set of structural

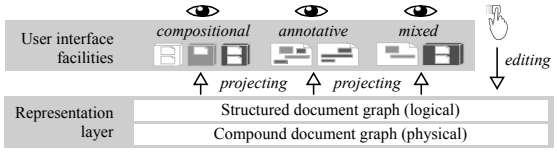

Figure 1: Global picture of the approach

elements. Each element either represents content or the appearance of a VP. For instance, a variability-aware *abstract syntax tree* (AST) is built of structural elements and VPs.

The core elements in the graph are modules, which support the implementation of fine-grained extensions. Developers start with a *base module*, which implements an initial and valid product of the product line. Any further module adds, removes, or replaces feature details. To enable the composition of modules, we implemented simple addition and subtraction algorithms operating on an SDG. Combining the two algorithms enables replacing feature details (cf. [3]).

Figure 1 depicts the global picture of our approach. On the representation layer, an SDG (our proposed logical representation) is mapped onto a *compound document graph* [16,17] (physical representation). In a nutshell, we embed variability directly into compound documents, which support a fine-grained reuse of content. Thus, we inherently avoid code clones related to multiple join points. As a result, we omit complicated workarounds on the tool level. For instance, it is conceivable to emulate SDA-conform structured documents by combining metaexpressions [6, 14] (enabling the fine granularity) and code clone controlling facilities [9, 24] (handling the code clones related to multiple join points).

The SDG can be employed to render different editable user interfaces, for instance, a compositional/annotative projection (cf. Fig 1). *Mixed projections* support the implementation of variability using a configurable mixture of separate modules and annotations. As a result, developers can choose the best projection for a given task and even move fluidly between projections during development. This paper lays the foundation for such projections and thus contributes to the objective of overcoming the current trade-off between modularity and expressiveness [5, 13].

The main contributions of this paper are the logical model of SDGs (i.e., an abstract syntax for a variant of SDA), and its evaluation in different scenarios. Thus, instead of discussing projectional details, we focus on model details (Sec. 4) and the algorithms that operate on SDGs to produce different products (Sec. 5). To evaluate our model, we conducted a series of case studies using a prototype (Sec. 6).

2. RELATED WORK

Integration of SPL Implementation Approaches. The desired integration of compositional and annotative approaches is supported by the *choice calculus*, a sound formal programming language [26]. Yet, the choice calculus is bound to its language. Our SDA-based approach is more general, as it supports to featurize arbitrary structured content.

Currently, there are only two approaches that put into practice the desired integration [12, 27]. Kästner et al. discuss the benefits of an integration [10], and on this basis present a model for refactoring annotations into physically separated code and vice versa [12]. However, physical separation is less expressive and thus refactoring necessitates workarounds such as hook methods [12] or code clones [22].

Walkingshaw and Ostermann build upon the choice calculus and propose a sound formal model of editable views on variability-aware programs [27]. Instead of hiding feature details on the tool level, they propose to generate different simplified editable documents from a variability-aware generic AST. After editing a document, the changes can be committed back into the AST in a consistent manner. Moreover, their approach allows adding, removing and replacing feature details. We share the idea of providing a common underlying representation with Walkingshaw and Ostermann. However, due to building upon the choice calculus, their approach is bound to a formal programming language. Moreover, they pursue a parser-based approach, while we work towards a projectional one (cf. [7, 25]).

We note that a full-fledged applied solution for the desired integration as defined in Section 1 is not available yet.

Other closely related approaches. CIDE mitigates the drawbacks of classical annotative approaches on the tool level [10]. For instance, CIDE emulates locality and cohesion by hiding feature code on demand. However, CIDE only allows adding feature details, but neither removing nor replacing them [27]. Moreover, CIDE does not support multiple join points. The key difference between our approach and CIDE is, that we embed variability directly into the representation layer instead of leveraging the tool level.

Delta-oriented programming (DOP) is a programming language approach to implement software product lines [21]. To add, remove and replace feature details, so-called *delta modules* change a *core module*, which contains the base code. Conceptually, the core module is close to our base module. Delta modules have an `after` clause, which specifies the composition order, and a `when` clause, which must be satisfied to apply a delta module. Thus, it may be difficult to reuse a single delta module in different settings (without replication), which in turn is inherently supported by SDA. The key difference is, that we embed variability into the representation and thus leave the concrete syntax of modules open to the different projections (cf. Fig. 1).

3. APPROACH—INFORMAL OVERVIEW

Before we describe our logical model for creating SDGs in detail, we give an idea of our approach and discuss examples.

3.1 SDA basics

Figure 2a depicts a simple example of a stack product line. The base module *Base* implements an initial valid product. In particular, *Base* assigns the fragment f_0, which contains the class `Stack`, to the variation point vp_0. Note that vp_1, vp_2, and vp_3 appear in f_0. These VPs are assigned by the modules *Log* and *Peek*. Moreover, vp_2 appears multiple times across the program and thus the corresponding fragment f_2 is a single extension applied to multiple join points.

Figure 2b shows a projection for the composition of *Base* and *Log* (i.e., vp_0, vp_1 and vp_2 are filled with content). The

projection (similar to a CIDE view [11]) is a visually annotated document: *Base* fragments are colored clear, *Log* fragments light grey and *Peek* fragments are projected out.

3.2 Structured document graph basics

In Figure 3, we illustrate the SDG for our stack product line (Fig. 2). Note that we represent the source code as an AST.

As required, the fragment f_0 contains the entire document (cf. Fig. 2a), with vp_0 being the entry point to this document. We say that f_0 *instantiates* vp_0 and, accordingly, vp_0 *can be filled by* f_0 with content. This content is a graph of structural elements and VPs, while, once again, the VPs can be filled by fragments (e.g., vp_1 can be filled by f_1). VPs can also appear multiple times across a fragment (or fragments). Thereby, connections to a VP are made from multiple structural elements. For instance, vp_2 is connected with two elements in f_0 and with one element in f_3.

Coloring denotes the association of fragments and modules (cf. Fig. 2a). For instance, f_1 is colored light gray and thus assigned to vp_1 by *Log*. We say that *Log administers* vp_1.

3.3 Alternative fragments in the graph

Conditional compilation with the CPP allows the implementation of multiple alternative fragments. Figure 4a depicts a simple example, where `#if`, `#elif` and `#else` conditions denote which alternative features can be chosen in a product configuration. So either the code of *X*, *Y*, or of any other feature is selected in the final product.

Our SDA-based approach is similar: each VP can be filled by multiple alternative fragments or, in other words, multiple fragments can instantiate the same VP. Note that a module can assign a VP only once, as the assignment needs to be unique. Figure 4b illustrates these ideas and the fact that multiple alternative fragments can fill vp_0.

The lattice depicted in Figure 4c illustrates the possible fillings of vp_0 in a product line configuration (cf. [3]). Either f_0, f_1, f_2 or any other fragment can fill vp_0 at a time. In the case of overspecification (i.e., multiple modules allocate the same VP at the same time), a special error fragment ($\lightning$) is assigned to vp_0. The fragment f_0 in turn denotes the *default fragment*, which is chosen by default, but can be replaced by an alternative ($f_1, f_2, ...$). Note that only the base module assigns the default fragment.

4. LOGICAL MODEL OF STRUCTURED DOCUMENT GRAPHS

In Figure 5, we propose a logical model of SDGs (i.e., an abstract syntax for SDA concepts). Note that the distinctions made in SDA introduced inevitable complexity to our logical model rendering it less elegant. However, the physical representation we employed for our prototype is less complex [16]. Moreover, we achieved our main objective: the model enables a common fine-grained representation for compositional and annotative approaches.

Coloring modules and fragments. A module is associated with multiple fragments. Each fragment in turn is associated with exactly one module. We illustrate this relationship by means of coloring modules and their fragments (cf. Fig. 2 and 3). Note that the color of a module is unique, and exactly one color can be assigned to a fragment. Feature interactions can be implemented in a module with a new color mixed from the colors of the interacting modules [3].

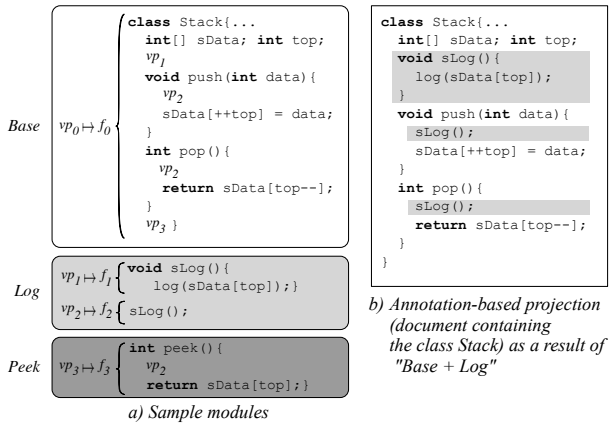

Figure 2: Modularization of a stack

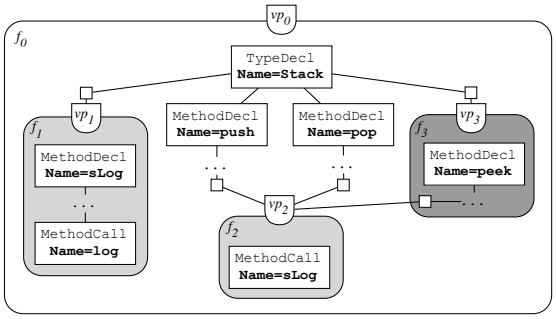

Figure 3: Structured document graph of stack

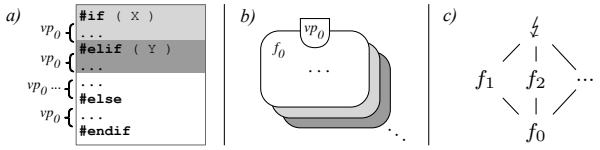

Figure 4: Multiple alternative fragments

Structuring the content of fragments. A fragment is the head of an ordered tree, whose nodes are structural elements and VPs. Note that our illustration in Figure 3 simplifies this idea. There, fragments are just containers for content (e.g., f_1 contains the method declaration `slog`).

According to our model, a structural element can have two different types of children: other structural elements or a VP. In Figure 3, the type declaration of the class `Stack` has four structural elements as children, while two of these children have a VP as a child (i.e., vp_1 and vp_3).

In contrast to structural elements, VPs may appear multiple times across a fragment (or fragments) and thus may have multiple parents (e.g., vp_2). The model enables tracing every appearance of a VP efficiently. We just need to

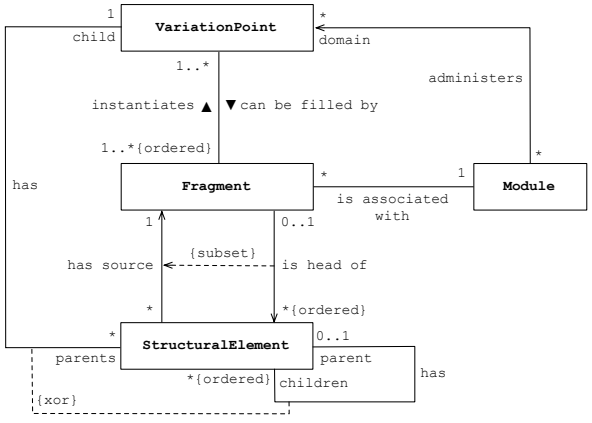

Figure 5: Model of structured document graphs

look up its parent structural element (or elements). Then, the *has-source* relation of the parents enables identifying the corresponding fragments. Thus, we know that vp_2 appears in f_0 and f_3 (cf. Fig. 2 and 3).

Filling variation points with content. A fragment can instantiate one or more VPs. Thus, multiple VPs can be filled with content from the same fragment. Note that this trait has the same effect to the program as the appearance of a VP across a document (or documents): a single extension is applied to multiple join points.

Moreover, a VP can be filled by multiple alternative fragments. In particular, a VP maintains an ordered list of these fragments (cf. Fig. 4). As the set of VPs instantiated by the fragments of a module needs to be unique [3], a module cannot assign the same VP multiple times.

Computing the domain of a module. To compute the domain $dom(m)$, which is the set of VPs a module m administers, we simply collect the VPs instantiated by the fragments associated with m. For instance, the module *Log* is associated with f_1 and f_2, which instantiate vp_1 and vp_2, respectively (i.e., $dom(Log) = \{vp_1, vp_2\}$). As we will see in the next section, the knowledge of the domain of a module comes in handy for product line configuration.

5. COMPOSING PRODUCTS

According to Figure 1, users see and interact with projections (also known from projectional editors [25]). Thereby, information can be "projected out" without making changes to the underlying graph.

To enable such conditional renderings (i.e., a generation of products), the algorithms construct a temporary module in the in-memory representation of the model. The fragments aggregated by this module are used to fill VPs with content in a final configuration step. In particular, each instance of a VP registers, which of its fragments (*can-be-filled-by* relation) provides the actual content. In other words, "can be filled by" turns into "is filled by" (cf. Fig. 5). As a result, every instance of a VP is either filled by exactly one instantiating fragment or left empty.

Note that a temporary module only knows about the fragments (i.e., there is no fragment associated with the temporary module in the graph). Thus, the internal representation of the SDG remains untouched.

In the following, we propose algorithms that operate on an SDG in order to add (Sec. 5.1), remove (Sec. 5.2), and replace (Sec. 5.3) feature details during a product configuration. Readers interested on algebraic properties are referred to SDA and corresponding formal definitions (not operating on an SDG) [3].

5.1 Module addition

By adding modules, we can add feature details [3]. A corresponding algorithm is straightforward. Let a and b be two modules, and $F(a)$ and $F(b)$ the corresponding sets of fragments associated with a and b, respectively. Then, we define the addition of modules as the union of their associated sets of fragments:

$$a + b =_{df} F(a) \cup F(b)$$

An algorithm for module addition assigns this resulting set of fragments to a temporary module m_{a+b}. Note that m_{a+b} only knows about the fragments of the added modules. Thus, in contrast to modules that appear in the graph, VPs in $dom(m_{a+b})$ may be instantiated multiple times. As desired, such a faulty configuration results in an error $\lightning$ (cf. Fig. 4).

Example. Given the example in Figure 2, we calculate $Base + Log$ as the union of their corresponding sets of fragments: $F(Base) \cup F(Log) = F(m_{Base+Log}) = \{f_0, f_1, f_2\}$. To render this product configuration, the corresponding VPs are filled with content by the fragments of the temporary module $m_{Base+Log}$. □

5.2 Module subtraction

By subtracting modules, we can remove feature details [3]. In particular, when two modules assign a fragment to the same VP (cf. Fig. 4), then subtraction provides the configuration decision that the feature detail of the minuend (i.e. the fragment's content) is not included in the final product. A simple subtraction algorithm operating on the SDG and giving a temporary module as a result, looks as follows:

```
1: function a − b
2:     instantiate m_{a−b} with a (⇒ F(m_{a−b}) = F(a))
3:     for each f ∈ F(m_{a−b}) do
4:         for each vp ∈ INSTANTIATES( f ) do
5:             if vp ∈ dom(b) then
6:                 F(m_{a−b}) \ {f}
7:     return m_{a−b}
```

In a nutshell, we compute and compare the domains of a and b. We start our subtraction algorithm by instantiating our temporary module m_{a-b} with a (Line 2). As a result, m_{a-b} knows all fragments associated with a. Next, we collect all fragments in $F(m_{a-b})$ (i.e., the ones in $F(a)$), and the VPs instantiated by these fragments (Lines 3 and 4).

Subsequently, we look up, whether any of these VPs is in the domain of b (Line 5) as well. If a VP is in $dom(b)$, then $dom(a) \cap dom(b) \neq \emptyset$ and we remove the corresponding fragments from our set of fragments of the temporary module $F(m_{a-b})$ (Line 6). Finally, we return m_{a-b} (Line 7).

Examples. Let m, n and p be three modules that assign fragments to VPs as follows:

$$m : \left\{ \begin{array}{l} vp_1 \mapsto f_3 \\ vp_3 \mapsto f_4 \\ vp_4 \mapsto f_5 \end{array} \right. , \quad n : \left\{ \begin{array}{l} vp_1 \mapsto f_1 \\ vp_2 \mapsto f_2 \end{array} \right. , \quad p : \{ \; vp_5 \mapsto f_6$$

To calculate $n - m$ and thus remove a feature detail provided with n (i.e., f_1), we use our subtraction algorithm. In particular, we instantiate the temporary module m_{n-m} with n. Subsequently, we remove f_1 from the set of associated fragments of m_{n-m} as $dom(n) \cap dom(m) = \{vp_1\}$. Consequently, m_{n-m} does not assign f_1 to vp_1. Instead, its partial function looks as follows: $m_{n-m} : \{vp_2 \mapsto f_2$.

Configuring a product line, we can also mix operations. For instance, to calculate $(n - m) + p$, we simply subtract m from n and then add p with $m_{(n-m)+p}$ as the resulting temporary module. These operations on the graph yield $F(m_{(n-m)+p}) = F(m_{(n-m)}) \cup F(p) = \{f_2\} \cup \{f_6\}$, and thus $m_{(n-m)+p}$ looks as follows:

$$m_{(n-m)+p} : \{vp_2 \mapsto f_2, vp_5 \mapsto f_6$$

For arbitrary modules m, n and p, we pick up the SDA subtraction laws as another example [3]. To show that both sides of the equations (e.g., $(m+p)-n = (m-n)+(p-n)$) yield the same temporary module is straightforward. The same holds for every other subtraction law (cf. [3]). □

5.3 Module overriding

Overriding enables to replace feature details. SDA defines overriding as a combination of module subtraction and addition [3]. In particular, overriding a module n by m is defined as follows:

$$m \twoheadrightarrow n =_{df} m + (n - m)$$

Thus, an implementation of $\twoheadrightarrow$ is straightforward using the aforementioned algorithms operating on an SDG.

Example. In the following, we compute $m \twoheadrightarrow n$ for the modules m and n as defined in the last section (5.2). There, our subtraction algorithm already gives the following partial function for the temporary module m_{n-m} as a result: $m_{n-m} : \{vp_2 \mapsto f_2$. Thus, we just calculate the sum of m and m_{n-m}. As $F(m) = \{f_3, f_4, f_5\}$ and $F(m_{n-m}) = \{f_2\}$, the resulting temporary module $m_{m \twoheadrightarrow n}$ looks as follows:

$$m_{m \twoheadrightarrow n} : \{vp_1 \mapsto f_3, vp_3 \mapsto f_4, vp_4 \mapsto f_5, vp_2 \mapsto f_2 \quad \square$$

6. IMPLEMENTATION & CASE STUDIES

We have implemented our model of SDGs on top of the representation layer of the *snippet compound document system* [16, 17], which is written in Objective-C for Mac OS X. There, content is structured using a compound document graph. The graph allows the fine-grained reuse of content in practice, but does not inherently support product line concepts. Introducing our SDG on top of this fine-grained layer enables the desired variability-awareness and the integration of compositional and annotative approaches (cf. Fig. 1).

To support the configurability of the SDG and thus the product line, we implemented the presented algorithms for module addition, subtraction and overriding. As a result, we were able to produce products that add, remove and replace feature details.

To demonstrate that our prototype can represent both implementation approaches, we took up a popular tool of each approach—FeatureHouse [2] (compositional) and CIDE [11] (annotative)—and imported corresponding examples. In the context of a Bachelor's [8] and a Master's thesis [20], we prototypically implemented the required transformations. The imports are similar to the refactorings described by Kästner et al. [12], with the difference that we transform the concrete syntax (and if necessary accompanying color information) into an SDG.

6.1 FeatureHouse examples

FeatureHouse separates features into directories and employs a so-called *feature structure tree* (FST) to represent arbitrary content. In a nutshell, FSTs are stripped-down ASTs, which can be composed using *superimposition* [2].

Import. During our tests, we focused on FeatureHouse projects written in Java. To import FeatureHouse projects into our prototype, we extended FeatureHouse in a way that made it possible for us to serialize FSTs and export the data as a JSON[1] array. In our prototype, we implemented a transformation that stores these FSTs in an SDG.

Transformation. To build the graph, we proceeded in the composition order specified in FeatureHouse. Thus, if composition order matters and has to be changed, there are two options: restructuring the graph (1) or changing the composition order in FeatureHouse and import again (2). We decided in favor of the latter option as the objective was just to see, whether we were able to produce valid products for a fixed composition order.

To represent FeatureHouse projects as SDGs, we employed an ad-hoc approach (to reduce development effort), which maintains the FST. Thereby, we roughly followed the mechanisms introduced by Kästner et al. to refactor compositional into annotative product lines: *copy flat*, *inline*, *merge* [12].

In the following, we give an example for two common use cases: method introduction and method refinement. Figure 6 illustrates an excerpt of the features *Base* and *Latches* in the Berkeley DB FeatureHouse example, and the corresponding SDG. In particular, the feature *Latches* introduces a new method (Line 2) and adds an assertion within the existing method `setroot` (Line 5). Then, with the keyword `original`, the original method specified in the *Base* feature is executed (Line 6).

The corresponding SDG (based on an AST) is simple. We just add a VP-fragment pair for the method declaration and the assertion (colored gray). As desired, the assertion is added before the original assignment.

Moreover, the example illustrates that compositional approaches lack expressiveness. The method parameter `notLatched` is added to the base code (Line 2). Thus, the feature *Latches* has not been modularized completely. Using the SDG, it is possible to assign the parameter `notLatched`

[1] http://json.org

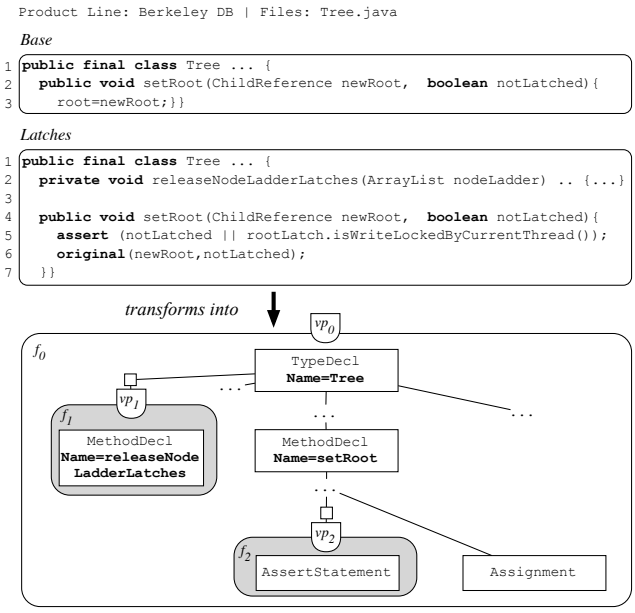

Figure 6: SDG for method introduction/refinement

to the feature *Latches*. Then, a mixed projection helps developers to identify that `notLatched` belongs to *Latches* and not *Base*.

Pretty Printing & Test. To test product configurations, we implemented a simple pretty printer. As a result, we were able to produce products using the SDG. Finally, we compared the results to the output generated by FeatureHouse. Among others, we repeatedly and randomly generated different product configurations of the following Java product lines, which are available as FeatureHouse superimposition examples:

Product line	LOC	Features
AJStats	15k	20
Bali2Jak	13k	11
Berkeley DB	64k	99
GPL	2k	26
Violet	10k	88

Table 1: Tested FeatureHouse Java examples[2]

Then, we used JPlag[3] to evaluate the equality of our generated code and the one generated by FeatureHouse. JPlag is a tool for plagiarism detection, which indicates the level of similarity of given source code files.

In almost every tested configuration, JPlag gave us 100% similarity and thus, generated programs were syntactically equal. In the rare cases of deviation, we manually inspected the distinctions. For instance, in a configuration of the *graph product line* (GPL) [18], we had a similarity of 98,9%. This deviation came from empty wrapper functions, which were never called, but generated by FeatureHouse. Our prototype in turn did not generate these empty functions. Also, some wrappers of single function calls were not generated by our prototype—instead, they were called directly.

Note that we did not implement a refactoring from the SDG back into FeatureHouse. This would not have provided any additional value, as the aim of the implementation was to show, that our proposed logical model is general enough to represent composition-based product lines. Moreover, we focused on Java programs and thus did not extend FeatureHouse to have it export other languages into our prototype. However, in FeatureHouse data is always represented as an FST and thus implementing such an export is a technical issue, not a conceptual one.

6.2 CIDE examples

CIDE is an Eclipse-based tool that supports to visually annotate product lines using different background colors, each of which represents a feature (cf. Fig. 2b). To store annotations, each code file comes with a corresponding XML-file, which maps features to AST nodes. CIDE provides a generator, which takes an annotated language grammar as input and generates a parser and pretty printer [15].

Import. To support the import of CIDE projects into our prototype, we implemented an Eclipse plugin for CIDE that handles the data exchange. In particular, we serialized the ASTs and the corresponding color files using JSON and exported them. In our prototype, we imported this data and mapped the ASTs to the SDG.

Transformation. As coloring of entire files and AST nodes already gave us the required variability information, mapping was mainly one-to-one and thus trivial. For instance, a colored method declaration such as `slog` depicted in Figure 2 is mapped to a corresponding VP-fragment pair as depicted in Figure 3.

Colored wrappers required special treatment [15]. To support wrappers, we followed the idea of using overriding [3]. Figure 7 illustrates a colored `try` statement taken as an example from the Berkeley DB. There the `try` statement as well as the complete `finally` block are colored. Thus, we have alternative pieces of code: either there is a base code block or a surrounded one. In particular, vp_1 reflects this alternative as it can be either filled by $f_{1.1}$ or $f_{1.2}$. The base code block attached to vp_2 in turn appears among these fragments. Then, *Latches*$\rightarrowtriangle$*Base* gives us the following set of fragments $\{f_0, f_2, f_{1.2}\}$, while solely selecting *Base* still returns a valid configuration. Moreover, *Base*+*Latches* returns the correct set as $f_{1.1}$ is a default (cf. Fig. 4c).

Export. As our approach is closely related to CIDE, we also implemented an export of the SDG into source code and XML files containing color information. On this occasion, we adapted the generator provided with CIDE to have it generate a pretty printer written in Objective-C, which operates on a SDG. We also used this printer to create products generated from an SDG.

Note that CIDE only provides an annotated grammar for Java 1.5. Thus, we were able to generate a pretty printer for product lines that can be tested under Java 1.5 only. For Java 1.6 and higher, CIDE uses the standard JDT parser and

[2]http://www.infosun.fim.uni-passau.de/spl/apel/fh/
[3]https://jplag.ipd.kit.edu

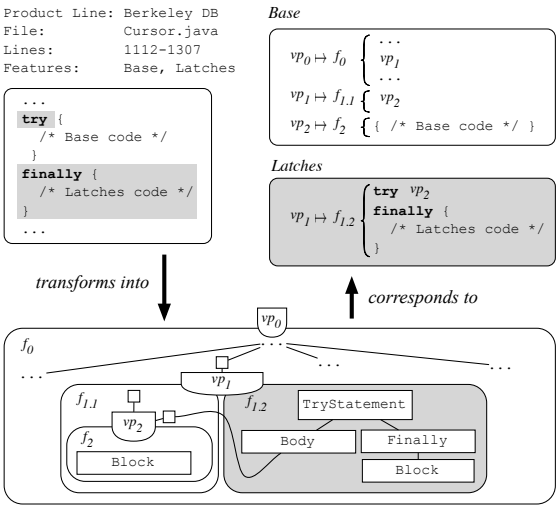

Figure 7: SDG for a colored wrapper

introduces a wrapper in the tool infrastructure to map the JDT AST into a CIDE compatible one.

Test. In summary, we imported, exported, and configured the CIDE example projects depicted in Table 2.

To test a product line with sufficient complexity written in Java, we took the GPL, which has been proposed as a standard problem for evaluating product-line methodologies [18]. We compared the imported and exported Java files as well as the produced products using JPlag, which always gave us 100% similarity. To compare the XML-files, where we also found no differences, we wrote a small script that pretty prints and diffs the files. We also wrote a script to compare the results of the Haskell Graph Library and the Berkeley DB documentation and could not find any differences.

Product line	Language	LOC	Features
GPL	Java	1k	21
Haskell Graph Library	Haskell	1k	18
Berkeley DB doc.	HTML	110k	42

Table 2: Tested CIDE examples[4]

7. THREATS TO VALIDITY

We imported and tested only examples provided with FeatureHouse and CIDE. Consequently, no evaluation for other tools/languages exists so far. However, our proposed concepts are independent of the concrete syntax and thus independent of a specific language. Due to the very nature of compositional and annotative approaches, we assume that our representation layer can be used to represent examples of other tools and languages as well.

We did not formally prove that the transformations during import and pretty printing preserve semantics. However, we randomly tested transformations by pretty printing multiple products using the graph (created during import) and comparing the results with the ones produced by the original tool. The products we produced were syntactically equivalent, except for rare cases, which we manually inspected for semantic differences (we did not find any; cf. Sec. 6.1). Thus, we assume that our transformations are correct.

For our tests, we carefully selected example product lines of different domains, size and complexity. However, the examples we have tested were implemented following the methodology of the respective tool. Consequently, as we did not modify the example projects, we were not able to test all features and operations provided with our approach in a larger scenario. For instance, CIDE only allows adding feature details. Thus, we were not able to test subtraction and overriding using the unchanged CIDE code base. Nevertheless, our intent was to show that our model is general enough to represent both implementation approaches for software product lines.

To test all operations, we implemented small test cases from scratch and produced products using the algorithms. As a result, we always composed correct programs, but still, a case study that tests the full potential of our approach in a larger setting is necessary. On this occasion, we plan to revise the GPL such that the capabilities of our approach are fully leveraged.

Moreover, we did not formally prove equivalence between our variant of SDA and SDA itself. However, we used small test cases to see whether our implementation meets the laws and algebraic properties specified by Batory et al. [3]. The results are promising. Due to space restrictions, we omitted a detailed discussion and thus left a formal prove as subject for future work.

So far, our results provide first insights and show the feasibility and applicability of our ideas.

8. CONCLUSION AND FUTURE WORK

In this paper, we focused on the representation of product lines and, on the basis of SDA, presented a logical model for variability-aware structured documents. The model enables to put into practice the integration of compositional and annotative approaches. Structured documents are represented as a graph of interrelated VPs, fragments, modules, and structural elements. We proposed algorithms that operate on such a structured document graph to add, remove and replace feature details. To evaluate our approach, we implemented the presented concepts, and imported and configured product lines of each feature-based implementation approach. For CIDE-based projects, we implemented an export back into CIDE.

With regard to future work, we plan to provide the desired editable projections (cf. Fig. 1). On this occasion, we are currently evaluating whether it is possible to implement our model using the MPS[5] language workbench, which already provides a rich set of features [7]. Based on our model, it might be possible in MPS to plug-in existing variability-aware languages and concepts (DOP, FeatureHouse, etc.) as well as designing new ones. Altogether, we assume that the

[4]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/

[5]http://jetbrains.com/mps

implementation of our model has the potential to serve as a generic framework for evaluating and comparing variability-aware implementation approaches.

In summary, providing a projectional approach on top of the generic variability-aware abstract syntax presented in this paper opens new opportunities for implementing software product lines from scratch. As the concrete syntax is independent of the abstract syntax provided with our model, different textual/visual projections are imaginable. As a result, developers are not required to explicitly type assignments of fragments to VPs as depicted in Figure 2a. Instead a developer can interact, for instance, with a coloring projection where VPs are implicit (cf. Fig. 2b). If a developer faces a limitation in the current projection, he can move fluidly to a different one that better suits the current task. Thus, instead of introducing additional editing complexity, our approach even offers a major benefit concerning the editing process.

9. ACKNOWLEDGMENTS

We thank our students Moritz Fey and Jochen Palz for their assistance on prototypically implementing the transformations for FeatureHouse and CIDE, respectively. Moreover, we thank both for their help with transforming the example product lines and the comparison of produced products.

10. REFERENCES

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

[2] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Eng.*, pages 63–79, 2013.

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

[4] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, pages 355–371, 2004.

[5] B. Behringer. Integrating approaches for feature implementation. In *FSE*, pages 775–778, 2014.

[6] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE*, pages 422–437, 2005.

[7] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. Evaluating and comparing language workbenches. *Computer Languages, Systems & Structures*, 44:24–47, Dec. 2015.

[8] M. Fey. Klassisch-modulare Software-Produktlinien im Snippet System (german). Bachelor's thesis, htw saar, 2015.

[9] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *ICPC*, pages 238–242, 2009.

[10] C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *McGPLE*, pages 35–40, 2008.

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

[12] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *GPCE*, pages 157–166, 2009.

[13] C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *SPLC*, pages 5:1–5:8, 2011.

[14] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM ToSEM*, 21(3):1–39, 2012.

[15] 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*, pages 175–194, 2009.

[16] L. Kirsch, J. Botev, and S. Rothkugel. Snippets and Component-Based Authoring Tools for Reusing and Connecting Documents. *JDIM*, pages 399–409, 2012.

[17] L. Kirsch, J. Botev, and S. Rothkugel. The Snippet Platform Architecture: Dynamic and Interactive Compound Documents. *ICIMT*, pages 161–167, 2013.

[18] R. E. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE*, pages 10–24, 2001.

[19] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, pages 169–194, 2005.

[20] J. Palz. Annotierter Source-Code im Snippet-System: Toolchain zur Transformation und Datenhaltung von CIDE-Projekten (german). Master's thesis, htw saar, 2015.

[21] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *SPLC*, pages 77–91, 2010.

[22] S. Schulze. *Analysis and Removal of Code Clones in Software Product Lines*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2013.

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

[24] R. D. Venkatasubramanyam, H. K. Singh, and K. Ravikanth. A method for proactive moderation of code clones in IDEs. In *IWSC*, pages 62–66, 2012.

[25] M. Völter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In *SLE*, pages 41–61, 2014.

[26] E. Walkingshaw and M. Erwig. A calculus for modeling and implementing variation. In *GPCE*, pages 132–140, 2012.

[27] E. Walkingshaw and K. Ostermann. Projectional editing of variational software. In *GPCE*, pages 29–38, 2014.