Recovering Traceability between Features and Code in Product Variants

Lukas Linsbauer
Johannes Kepler University
Linz, Austria
k0956251@students.jku.at

Roberto E. Lopez-Herrejon
Johannes Kepler University
Linz, Austria
roberto.lopez@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Many companies offer a palette of similar software products though they do not necessarily have a Software Product Line (SPL). Rather, they start building and selling individual products which they then adapt, customize and extend for different customers. As the number of product variants increases, these companies then face the severe problem of having to maintain them all. Software Product Lines can be helpful here - not so much as a platform for creating new products but as a means of maintaining the existing ones with their shared features. Here, an important first step is to determine where features are implemented in the source code and in what product variants. To this end, this paper presents a novel technique for deriving the traceability between features and code in product variants by matching code overlaps and feature overlaps. This is a difficult problem because a feature's implementation not only covers its basic functionality (which does not change across product variants) but may include code that deals with feature interaction issues and thus changes depending on the combination of features present in a product variant. We empirically evaluated the approach on three non-trivial case studies of different sizes and domains and found that our approach correctly identifies feature to code traces except for code that traces to multiple disjunctive features, a rare case involving less than 1% of the code.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Algorithms, Theory

Keywords

Product Variants, Features, Traceability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPLC 2013, August 26-30, 2013, Tokyo, Japan
Copyright 2013 ACM 978-1-4503-1968-3/13/08 ...$15.00.

1. INTRODUCTION

It is common practice for companies to sell variants of a software product, typically to support different customer needs. These product variants share code (often quite extensively) but they also differ from one another and exhibit unique features or feature combinations not shared with others (i.e., to cover specific customer needs). Unfortunately, often such product variants are not developed and maintained as Software Product Lines (SPLs) [17]. Instead, these companies tend to build separate product variants where a new product variant is typically combined from various existing variants. This phenomenon has been observed in many domains including commercial partners in the avionics and steel manufacturing domain we work with. Our experience tells us that companies rarely build a common code base for these product variants with the ability to switch features on or off [9]. The reasons are at times technical (e.g., it may not be possible to compile a single product for different operating systems), are based on economic factors (e.g., code as an intellectual property is not made available unless paid for), or may even be mandated (e.g., certifying a safety critical system with dead – aka switched-off code – is undesirable). Whatever the reasons, these situations all have in common that variations of software products exist which are created and maintained separately.

Separately maintaining variations of (similar) product variants poses unique challenges. Bug fixes that affect shared features may need to be replicated across multiple products and, even more challenging, a feature may undergo evolutionary changes with successive product variants. In addition to the feature's functionality, software engineers must also define how the feature is supposed to interact with other features (or not to interact) in an adaption of the feature interaction problem. And they must do so separately in every product variant. We observed that maintaining product variants works well for a limited number of variants but as this number increases so does the complexity of maintaining the variants. This is the point where we observe companies to express interest in a transition to SPLs.

The traditional view of SPLs is a forward-engineering one where we first decide on possible features and variants that we may want to support. SPLs can also be helpful in a reverse-engineering manner as a means of maintaining the commonalities and differences among existing product variants. In earlier work, we demonstrated how to compute feature models from product variants [8]. This work lays the foundation for building a shared code base for these product variants - by computing the traceability between

features and code. This step is necessary because companies typically only know which features are implemented in which product variants but they do lack precise knowledge where in the code these features are implemented.

The specific goal of this paper is thus to establish the traceability between features and code with the additional challenge of distinguishing the part of a feature that is the same in every product where the feature appears versus the parts of a feature that change depending on the combination of features present in a product (i.e., the code that implements feature interactions). Our algorithm determines traceability by matching code overlaps and feature overlaps between product variants. The precision of the algorithm thus depends on the ability to distinguish individual features. Fortunately, we do not require a product variant for every possible feature combination. Rather, to distinguish any two features, we typically only require one product with and one without the feature. By matching these sets, we obtain code fragments that belong to individual features or groups thereof. Only in cases where features always occur together are we unable to exactly distinguish what code fragment belongs to which feature individually; however, even here we can still determine the traceability of both features together. Naturally, this implies that the precision of our algorithm is dependent on the number of product variants available (because then it is increasingly likely we get all necessary feature combinations).

As a first step we implemented our approach on the code granularity level of class methods and fields to demonstrate both scalability and correctness. We argue that this approach in principle also works on other levels of granularity, e.g. statements. We evaluated our approach on three case studies where we recovered traceability among 11-14 features and between 600-21,000 code fragments. Out of all these code fragments, less than 1% could not be traced correctly, in some cases because there were mistakes in the source code of the case studies (e.g. dead code that should have been removed from a product) and in others because code traced to multiple disjunctive features.

2. BACKGROUND AND EXAMPLE

The driving goal of our work is to provide support for maintaining collections of related product variants or for reverse-engineering such product variants into a software product line by automatically extracting traceability information between features and source code. We start off with a set of programs which we assume are currently maintained. Therefore for each program we initially know the functionality it provides and hence the features that it contains and its source code. By comparing these programs with each other we derive associations between parts of the source code and features or feature interactions. In this approach we assume that products which have common features also have common code, and that this common code implements exactly these common features. The automated traceability extraction is thus currently limited to code pieces with a unique trace as will be discussed in sections 3 and 5. As a first step, the focus of our work was at the code granularity level of classes, methods and fields. The results we obtained at this level encourage us to tackle granularity below this level as part of our future work.

c_1	`Point Line.startPoint`
c_2	`Point Line.endPoint`
c_3	`void Line.paint(Graphics g)`
c_4	`Line.Line(Point start)`
c_5	`void Line.setEnd(Point end)`
c_6	`Line.Line(Color color, Point start)`
c_7	`List Canvas.lines`
c_8	`void Canvas.wipe()`
c_9	`void Canvas.setColor(String colorString)`

Figure 1: Unique Code Pieces

2.1 Example: Draw Product Line (DPL)

The example we use to illustrate our approach consists of four product variants. We will refer to the SPL formed with these four variants as the *Draw Product Line (DPL)*. DPL variants are simple drawing programs with different capabilities – depending on the selected features of the product – such as drawing lines and rectangles, wipe the drawing area clean, or select a color to draw with. Their feature combinations and code are shown in this section along with the necessary background to understand the underlying theoretical model of our work.

A *feature list* FL is a set containing all the features available in a certain domain. In our example, we assume a feature list $\texttt{FL} = \{\texttt{LINE}, \texttt{WIPE}, \texttt{COLOR}\}$. A *feature set* of a product is a 2-tuple $(sel, \overline{sel})$ where sel is the set of features that are selected in the product and $\overline{sel}$ is the set of features that are not selected. The products' feature sets can also be described in form of a feature set table as shown in Table 1, there is a row for each product and a column for every feature in the SPL. If a product provides a feature there is a mark in the corresponding field. In other words, each row represents the feature set for the corresponding product [8].

The code snippets for the example products we will use are shown in Code Listing 1. For each of the four products the classes `Line` and `Canvas` are shown. $\texttt{Product}_1$ is from Line 1 to Line 11 and $\texttt{Product}_2$ from Line 13 to Line 22. The class `Line` looks the same for both. But in the class `Canvas` there is no method `wipe` for $\texttt{Product}_2$ as the feature `WIPE` is not part of it. $\texttt{Product}_3$ is from Line 24 to 34. The constructor of class `Line` at Line 28 now also contains a parameter for the color. The old constructor was removed. And the class `Canvas` contains a method for setting the color at Line 33. $\texttt{Product}_4$ is from Line 36 to Line 47 and is the same as $\texttt{Product}_3$ with the only difference that in class `Canvas` at Line 46 the method `wipe` is present again. This code base gives us a number of unique code pieces to work with. A code piece can either be a method or a field. They are summarized in Figure 1.

The products can also be described via feature algebra as presented in the work of *Feature Oriented Software Development (FOSD)* [12]. We use this algebraic notation to

Products	Wipe	Line	Color	Feature Sets
Product 1	✓	✓		$(\{\texttt{W},\texttt{L}\},\{\texttt{C}\})$
Product 2		✓		$(\{\texttt{L}\},\{\texttt{W},\texttt{C}\})$
Product 3		✓	✓	$(\{\texttt{L},\texttt{C}\},\{\texttt{W}\})$
Product 4	✓	✓	✓	$(\{\texttt{W},\texttt{L},\texttt{C}\},\{\})$

Table 1: Feature Set Table

```
1 /* Product 1 (WIPE, LINE) */
2 class Line {
3    Point startPoint, endPoint;
4    void paint(Graphics g) {...}
5    Line(Point start) {...}
6    void setEnd(Point end) {...}
7 }
8 class Canvas {
9    List lines = new LinkedList();
10   void wipe() {...}
11 }
12
13 /* Product 2 (LINE) */
14 class Line {
15   Point startPoint, endPoint;
16   void paint(Graphics g) {...}
17   Line(Point start) {...}
18   void setEnd(Point end) {...}
19 }
20 class Canvas {
21   List lines = new LinkedList();
22 }
23
24 /* Product 3 (COLOR, LINE) */
25 class Line {
26   Point startPoint, endPoint;
27   void paint(Graphics g) {...}
28   Line(Color color, Point start) {...}
29   void setEnd(Point end) {...}
30 }
31 class Canvas {
32   List lines = new LinkedList();
33   void setColor(String colorString)
         {...}
34 }
35
36 /* Product 4 (COLOR, WIPE, LINE) */
37 class Line {
38   Point startPoint, endPoint;
39   void paint(Graphics g) {...}
40   Line(Color color, Point start) {...}
41   void setEnd(Point end) {...}
42 }
43 class Canvas {
44   List lines = new LinkedList();
45   void setColor(String colorString)
         {...}
46   void wipe() {...}
47 }
```

Code Listing 1: DPL product snippets

describe our traceability mining algorithm more precisely. The idea is that a *product* can be composed by adding *features*. Features are written in uppercase letters. For example $\texttt{Product}_1$ is composed by extending feature LINE with feature WIPE, written as WIPE(LINE). This is called a *feature expression*.

Each *feature* consists of *modules*, which represent its implementation. The *base module* [12] of a feature contains the code that is always present in a product that has this feature, independent of any other features that may or may not be present. However, a module cannot add code that is already added by another module. This means a piece of code is assumed to be unique to a module, i.e. every piece of code has a unique trace. A base module is denoted in lowercase letters. For example, the base module of feature LINE is denoted as line. It contains the code of class *Line* that is always there if the feature LINE is selected. This means the constructor is not part of the base module, because it changes depending on other selected features. All products have the feature LINE, but not all have the same constructor in class *Line*.

The feature WIPE in $\texttt{Product}_1$ also consists of another module $\delta\texttt{line}/\delta\texttt{WIPE}$ which is called a *derivative module* [12]. It contains the changes the feature WIPE makes to the module line. In case of $\texttt{Product}_1$ there is no source code associated with that module and there are no changes. A change can be the addition of code as well as the alteration or removal of code. This interpretation is different to the one presented in [12] in that we allow the removal of code as will be explained in Section 3.3. Derivative modules basically model the interaction of features, how features influence each other. For example, only the combination of a number of specific features makes a certain piece of code necessary (or respectively unnecessary). There are also higher order derivatives like $\delta^2\texttt{color}/\delta\texttt{LINE}\delta\texttt{WIPE}$ in $\texttt{Product}_4$. It represents the changes that features LINE and WIPE make to module color. So the principle is the same, there are just more features involved. This example models the interaction of three features.

There are *two operations* on modules that allow us to compose them [12]. The first operation is $+$ which is a binary operation that unifies the code of two base modules. The code of two base modules is disjoint. The second operation is $\bullet$ which either composes two derivative modules into a composite derivative module or weaves the changes of a derivative module into a base module yielding a so called *woven base module*.

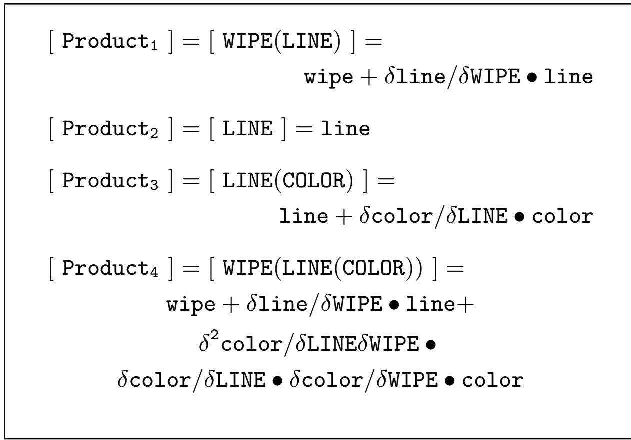
With these two operations the relationship between a *feature expression* and the corresponding *module expression* that implements it can be defined as shown in Figure 2 for the four products of Table 1. For example, $\texttt{Product}_3$ is composed by applying feature LINE to feature COLOR as can be seen in its feature expression. The product is implemented by the respective base modules line and color which contain the code that is always present for these features. In addition the changes that feature LINE makes to the base module color are woven into module color in the form of the derivative $\delta\texttt{color}/\delta\texttt{LINE}$.

As can be seen the module expressions grow very fast with the number of features. A product with n features has a feature expression consisting of n features and a module expression consisting of $2^n - 1$ modules. Notice that modules are separated by operations $+$ or $\bullet$.

Now we have our examples set up. We know for each of our example products their features, their module expressions and their source code. The ideal solution to the stated problem would be to know for each piece of source code to which module it belongs and the pieces of source code each module consists of. In practice however, it will not be possible to isolate every module. There may be modules for which their source code could not be distinguished, for example because two modules never exist without each other.

3. TRACEABILITY MINING ALGORITHM

In this section we present an algorithm for tracing products' features in their source code. The algorithm expects as input a number of products with their corresponding fea-



$[\ \texttt{Product}_1\ ] = [\ \texttt{WIPE(LINE)}\ ] =$
$\texttt{wipe} + \delta\texttt{line}/\delta\texttt{WIPE} \bullet \texttt{line}$

$[\ \texttt{Product}_2\ ] = [\ \texttt{LINE}\ ] = \texttt{line}$

$[\ \texttt{Product}_3\ ] = [\ \texttt{LINE(COLOR)}\ ] =$
$\texttt{line} + \delta\texttt{color}/\delta\texttt{LINE} \bullet \texttt{color}$

$[\ \texttt{Product}_4\ ] = [\ \texttt{WIPE(LINE(COLOR))}\ ] =$
$\texttt{wipe} + \delta\texttt{line}/\delta\texttt{WIPE} \bullet \texttt{line} +$
$\delta^2\texttt{color}/\delta\texttt{LINE}\delta\texttt{WIPE} \bullet$
$\delta\texttt{color}/\delta\texttt{LINE} \bullet \delta\texttt{color}/\delta\texttt{WIPE} \bullet \texttt{color}$

Figure 2: Products in Feature Algebra

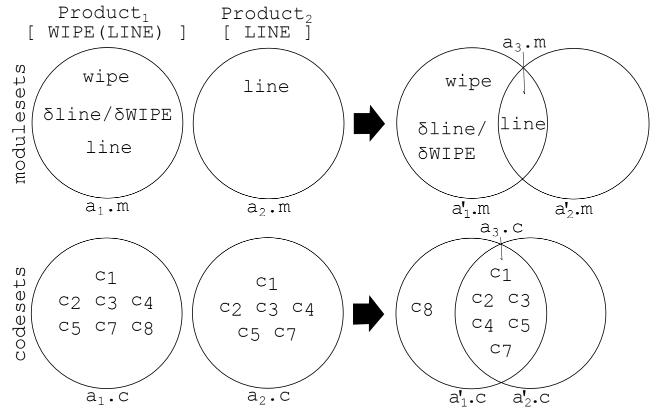


Figure 3: Intersection of Product 1 and Product 2

ture sets and code. For example, $\texttt{Product}_1$ with features LINE and WIPE. This product consists of only two features as its feature expression WIPE(LINE) shows. The corresponding module expression however consists of three modules: the base modules line and wipe and the derivative module $\delta\texttt{line}/\delta\texttt{WIPE}$. Each of these modules consists of its own code (if any). The algorithm then computes what code belongs to which modules.

3.1 Basic Insight

The general idea behind this algorithm is the observation that products that have features in common will also have code in common and vice versa. A *product* is defined as a tuple where the first element is its feature set, accessed via product.f, and the second element is the set of code it contains, accessed via product.c:

$$\texttt{product} = (\texttt{featureset}, \texttt{codeset})$$

Note that it is not sufficient to just look at a product's features as the above example shows, because the interactions between features, which are exactly the derivatives, would be left out. So instead of looking at products' features in their feature expressions we look at the modules in their module expressions.

The modules that two input products have in common are then associated with the code these same two products have in common. Assume $\texttt{Product}_1$ and $\texttt{Product}_2$ as input. $\texttt{Product}_2$ consists of only one module, which is the base module line. The code these two products have in common is therefore associated with the base module line. The remaining code in $\texttt{Product}_1$ is left for the base module wipe and the derivative $\delta\texttt{line}/\delta\texttt{WIPE}$.

The basic concept behind this is to treat products as *associations* between modules and code. We define an association to be a tuple where the first element is the set of modules and the second element is the set of code:

$$\texttt{association} = (\texttt{moduleset}, \texttt{codeset})$$

We access an association's moduleset via association.m and its codeset via association.c. For $\texttt{Product}_1$ and $\texttt{Product}_2$ these associations initially are:

$$\texttt{association}_1 = \texttt{a}_1 = (\ \{\texttt{line}, \texttt{wipe}, \delta\texttt{line}/\delta\texttt{WIPE}\}\ ,\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7, \texttt{c}_8\}\ )$$

$$\texttt{association}_2 = \texttt{a}_2 = (\ \{\texttt{line}\}\ ,\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\}\ )$$

Associations are then intersected by respectively intersecting their module sets and their code sets (see Figure 3). By doing so, it is possible to obtain new associations that are added to the list or alter existing associations. For example, after intersecting these two associations the set of associations is:

$$\texttt{association}'_1 = \texttt{a}'_1 = (\texttt{a}_1.\texttt{m} \setminus \texttt{a}_2.\texttt{m}, \texttt{a}_1.\texttt{c} \setminus \texttt{a}_2.\texttt{c}) = (\ \{\texttt{wipe}, \delta\texttt{line}/\delta\texttt{WIPE}\}\ ,\ \{\texttt{c}_8\}\ )$$

$$\texttt{association}'_2 = \texttt{a}'_2 = (\texttt{a}_2.\texttt{m} \setminus \texttt{a}_1.\texttt{m}, \texttt{a}_2.\texttt{c} \setminus \texttt{a}_1.\texttt{c}) = (\ \{\}\ ,\ \{\}\ )$$

$$\texttt{association}_3 = \texttt{a}_3 = (\texttt{a}_1.\texttt{m} \cap \texttt{a}_2.\texttt{m}, \texttt{a}_1.\texttt{c} \cap \texttt{a}_2.\texttt{c}) = (\ \{\texttt{line}\}\ ,\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\}\ )$$

The existing associations are altered by removing the elements in the intersection from their modules and code. The intersection is added as new $\texttt{association}_3$. For example, code $\texttt{c}_8$ which corresponds to method wipe in class Canvas (see Figure 1) is now associated with modules wipe and $\delta\texttt{line}/\delta\texttt{WIPE}$ in association $\texttt{a}'_1$ and code $\texttt{c}_1$ which corresponds to field startPoint in class Line (see Figure 1) is associated with the module line in association $\texttt{a}_3$ as displayed in Figure 3. This intersection process is repeated as new product sets are integrated. At the end, every feature and every piece of code appear exactly once. This means that a piece of code can only belong to a single module. Therefore our algorithm has the following limitation:

Unique Trace Limitation. *A piece of code that does not have a unique trace, i.e. traces to multiple modules, will either: i) not be assigned to any module, ii) be assigned to only one of the modules, or iii) be assigned to another module, depending on the configurations of product variants used as input.*

3.2 From Features to Modules

The algorithm starts with just the features of each product, so it needs to calculate the modules. A product with a set of n features has a module expression with $2^n - 1$ modules. The modules are obtained by building the powerset of

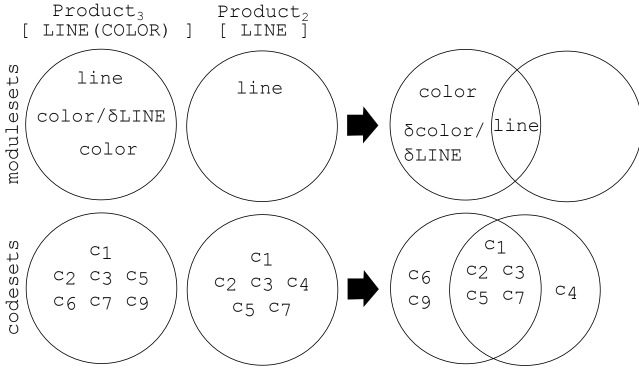


Figure 4: Intersection of Product 3 and Product 2

the set of features without the empty set:
$\texttt{moduleset} = \mathcal{P}(\texttt{featureset.sel})\backslash\{\emptyset\}$. For $\texttt{Product}_1$:

$$\texttt{moduleset} = \mathcal{P}(\{\texttt{LINE}, \texttt{WIPE}\})\backslash\{\emptyset\} = \\ = \{\{\texttt{LINE}\}, \{\texttt{WIPE}\}, \{\texttt{LINE}, \texttt{WIPE}\}\}$$

Sets with exactly one feature represent the base modules and sets with more than one feature (e.g. $\{\texttt{LINE}, \texttt{WIPE}\}$) represent the derivatives. There is no order in a set, so $\delta\texttt{line}/\delta\texttt{WIPE}$ has to be equal to $\delta\texttt{wipe}/\delta\texttt{LINE}$ in order for this to be legitimate. For our algorithm it does not matter in what order the features are added, it is only important to know whether two features interact or not, so we can easily assume that is the case.

$$\{\texttt{LINE}, \texttt{WIPE}\} = \delta\texttt{line}/\delta\texttt{WIPE} = \delta\texttt{wipe}/\delta\texttt{LINE}$$

3.3 Dealing with Code Removal

An limitation for this approach so far is that code can only be added. Modules are not allowed to remove code. So we have to find another way to model such effect. Assume we have a base module `line` which adds some pieces of code $\{c_1, c_2, c_3, c_4, c_5, c_7\}$ to a program. Then we add the feature `COLOR`, and therefore the modules `color` and $\delta\texttt{line}/\delta\texttt{COLOR}$, to the program. Module `color` adds some code just as module $\delta\texttt{line}/\delta\texttt{COLOR}$ does. But $\delta\texttt{line}/\delta\texttt{COLOR}$ also removes code c_4 (the old constructor) from the program. At this point the question arises whether code c_4 was part of the base module `line` to begin with. If it is not always present when the module `line` is present, then it is obviously not a part of it. But where else would it belong?

This is actually exactly what happens with the examples $\texttt{Product}_2$ and $\texttt{Product}_3$. Their intersection is shown in Figure 4. One can see, that code c_4 has no corresponding modules. The intersection of the modules at that point is empty.

Indeed, in a situation like this the code could be left over without any modules to associate it with. It also should be noted that this problem only applies to derivatives, as base modules could not remove code from a module other than itself, which would make no sense.

Negative Features. Instead of having a derivative remove a piece of code c_4 we would rather have another derivative add this same piece of code. But it does not fit into any of our derivatives we have so far. So we introduce *negative features*. For every feature `F` there is now also its negation $\neg\texttt{F}$. This leaves us with many new derivatives to work with. We can now associate c_4 with the derivative $\delta\texttt{line}/\delta\neg\texttt{COLOR}$.

So instead of having $\delta\texttt{line}/\delta\texttt{COLOR}$ remove c_4 we are now having $\delta\texttt{line}/\delta\neg\texttt{COLOR}$ add c_4. We have yet to define what exactly this means though and how we want to interpret such modules containing negative features. For example, $\delta\texttt{line}/\delta\neg\texttt{COLOR}$ could be interpreted as the derivatives of line and anything that is not color, but that is not what we want. Such an interpretation would just be an abbreviation for a list of other derivatives. It could also be interpreted as a synonym for the one derivative of line and all features except color. But this is also not what we want. We want $\delta\texttt{line}/\delta\neg\texttt{COLOR}$ to be its own module, with its own unique code, that does not have anything to do with other modules. And that is exactly how we interpret and use these modules.

With negative features, the feature and module expressions of our products now look different (see Figure 5). Each feature expression now contains every feature in the feature list exactly once, either positive or negative. Therefore each product consists of $2^N - 1$ modules now, where N is the number of features in the whole feature list (not just the features that are implemented by the product as before), only that modules containing only negative features/modules can be left out. Negative features/modules only make sense as a derivative with at least one positive feature/module.

$$[\ \texttt{Product}_1\ ] = [\ \neg\texttt{COLOR}(\texttt{WIPE}(\texttt{LINE}))\ ] = \\ (\neg\texttt{color}) + \delta\texttt{wipe}/\delta\neg\texttt{COLOR} \bullet \texttt{wipe} + \\ \delta^2\texttt{line}/\delta\texttt{WIPE}\delta\neg\texttt{COLOR} \bullet \\ \delta\texttt{line}/\delta\neg\texttt{COLOR} \bullet \delta\texttt{line}/\delta\texttt{WIPE} \bullet \texttt{line}$$

$$[\ \texttt{Product}_2\ ] = [\ \neg\texttt{WIPE}(\neg\texttt{COLOR}(\texttt{LINE}))\ ] = \\ (\neg\texttt{wipe} + \delta\neg\texttt{color}/\delta\neg\texttt{WIPE} \bullet \neg\texttt{color}) + \\ \delta^2\texttt{line}/\delta\neg\texttt{WIPE}\delta\neg\texttt{COLOR} \bullet \\ \delta\texttt{line}/\delta\neg\texttt{COLOR} \bullet \delta\texttt{line}/\delta\neg\texttt{WIPE} \bullet \texttt{line}$$

$$[\ \texttt{Product}_3\ ] = [\ \neg\texttt{WIPE}(\texttt{LINE}(\texttt{COLOR}))\ ] = \\ (\neg\texttt{wipe}) + \delta\texttt{line}/\delta\neg\texttt{WIPE} \bullet \texttt{line} + \\ \delta^2\texttt{color}/\delta\texttt{LINE}\delta\neg\texttt{WIPE} \bullet \\ \delta\texttt{color}/\delta\texttt{LINE} \bullet \delta\texttt{color}/\delta\neg\texttt{WIPE} \bullet \texttt{color}$$

$$[\ \texttt{Product}_4\ ] = [\ \texttt{WIPE}(\texttt{LINE}(\texttt{COLOR}))\ ] = \\ \texttt{wipe} + \delta\texttt{line}/\delta\texttt{WIPE} \bullet \texttt{line} + \\ \delta^2\texttt{color}/\delta\texttt{LINE}\delta\texttt{WIPE} \bullet \\ \delta\texttt{color}/\delta\texttt{LINE} \bullet \delta\texttt{color}/\delta\texttt{WIPE} \bullet \texttt{color}$$

Figure 5: Products in Feature Algebra with negative Features

For example, taking a closer look at $\texttt{Product}_3$ in Figure 5. The feature expression now contains one more feature, namely $\neg\texttt{WIPE}$. It is now explicit that the feature `WIPE` is not implemented by this product. This also reflects in the corresponding module expression. The first module $\neg\texttt{wipe}$ in parenthesis can be omitted as it is negative and does not interact with any positive features or modules. The second module $\delta\texttt{line}/\delta\neg\texttt{WIPE}$ however makes sense, as the implementation of the positive feature `LINE` can be influenced by feature `WIPE` not being present. In this case the *base module* `line` contains code that is always present if feature `LINE` is

present. In addition, the module δline/$\delta\neg$WIPE adds code that is specific to the implementation of feature LINE if feature WIPE is not present.

3.4 Algorithm Pseudo-Code

The following helper functions are used in the algorithm:

- *NOT(featureset)*: Negates all features contained in the set. For example:
  $NOT(\{\texttt{LINE}, \texttt{WIPE}\}) = \{\neg\texttt{LINE}, \neg\texttt{WIPE}\}$.
- *POW(featureset)*: Generates the powerset of the set (without the empty set and without modules consisting only of negative features/modules). This basically generates the modules for a set of features. For example:
  $\texttt{POW}(\{\texttt{LINE}, \neg\texttt{WIPE}\}) = \{\{\texttt{LINE}\}, \{\texttt{LINE}, \neg\texttt{WIPE}\}\}$.

The first part of the algorithm prepares all the input products for processing by converting them into initial associations. Each of these associations obtains its code from the corresponding product. The modules are calculated as the powerset of the union of the features of the product and the negated version of the features not contained in the product. The second part of the algorithm does the actual processing. One initial association after the other is processed and new associations are added to the final list of associations to be returned. The algorithm is shown in pseudo code in Algorithm 1. Some optimizations like not adding empty associations or merging associations that contain modules but no code are not shown to keep it simple.

Assume $\texttt{Product}_1$ and $\texttt{Product}_2$ as input for this algorithm. From line 5 to 14 the initial associations and data structures are prepared as follows:

$$\texttt{association}_1 = \left(\{\texttt{line}, \texttt{wipe}, \delta\texttt{line}/\delta\texttt{WIPE}, \delta\texttt{line}/\delta\neg\texttt{COLOR}, \delta\texttt{wipe}/\delta\neg\texttt{COLOR}, \delta^2\texttt{line}/\delta\texttt{WIPE}\delta\neg\texttt{COLOR}\}, \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7, \texttt{c}_8\} \right)$$

$$\texttt{association}_2 = \left(\{\texttt{line}, \delta\texttt{line}/\delta\neg\texttt{WIPE}, \delta\texttt{line}/\delta\neg\texttt{COLOR}, \delta^2\texttt{line}/\delta\neg\texttt{WIPE}\delta\neg\texttt{COLOR}\}, \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\} \right)$$

$$\texttt{init_assocs} = \{\texttt{association}_1, \texttt{association}_2\}$$

$$\texttt{associations} = \{\}$$

We start with an empty set associations to be returned by the algorithm and add associations as we iterate over init_assocs at line 16. Each initial association is intersected with every association in the associations set. We start with $\texttt{association}_1$. As associations is still empty there are no associations to intersect it with, the loop at line 19 is not entered. The remainder, which is equal to $\texttt{association}_1$, is added to the set as it is at lines 33 and 34.

$$\texttt{remainder} = \texttt{result}_1 = \texttt{association}_1{}^1$$

$$\texttt{associations} = \{\texttt{result}_1\}$$

[1] $\texttt{result}_x$ and $\texttt{association}_x$ are auxiliary variables to explain the algorithm, they do not appear in the code.

Algorithm 1 Traceability Mining Algorithm

```
1  Input: A List of Products (products),
2         A List of all Features (FL)
3  Output: A List of Associations (associations)
4
5  {convert products into initial associations}
6  init_assocs := {}
7  for p in products begin
8     association := (
9        POW(p.f.sel ∪ NOT(FL \ p.f.sel)),
10       p.c
11    )
12    init_assocs := init_assocs ∪ {association}
13 end
14 associations := {}
15 {iteratively process initial associations}
16 for a in init_assocs begin
17    remainder := (a.m, a.c)
18    new_assocs := {}
19    for a2 in associations begin
20       intersection := (
21          remainder.m ∩ a2.m,
22          remainder.c ∩ a2.c
23       )
24       remainder := (
25          remainder.m \ a2.m,
26          remainder.c \ a2.c
27       )
28       {alter existing association}
29       a2.m := a2.m \ intersection.m
30       a2.c := a2.c \ intersection.c
31       new_assocs := new_assocs ∪ {intersection}
32    end
33    associations := associations ∪ new_assocs ∪
34                    {remainder}
35 end
36 return associations
```

The next association to be processed is $\texttt{association}_2$. It is intersected with $\texttt{result}_1$ from line 20 to 31, as it is now in the associations set resulting in the following new associations:

$$\texttt{result}_1 = \left(\{\texttt{wipe}, \delta\texttt{line}/\delta\texttt{WIPE}, \delta\texttt{wipe}/\delta\neg\texttt{COLOR}, \delta^2\texttt{line}/\delta\texttt{WIPE}\delta\neg\texttt{COLOR}\}, \{\texttt{c}_8\} \right)$$

$$\texttt{remainder} = \texttt{result}_2 = \left(\{\delta\texttt{line}/\delta\neg\texttt{WIPE}, \delta^2\texttt{line}/\delta\neg\texttt{WIPE}\delta\neg\texttt{COLOR}\}, \{\} \right)$$

$$\texttt{intersection} = \texttt{result}_3 = \left(\{\texttt{line}, \delta\texttt{line}/\delta\neg\texttt{COLOR}\}, \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\} \right)$$

$\texttt{result}_1$ is altered (at lines 29 and 30) and remainder and intersection are added as new results (at lines 33 and 34).

$$\texttt{associations} = \{\texttt{result}_1, \texttt{result}_2, \texttt{result}_3\}$$

As there are no more initial associations the algorithm is done and the associations set is returned as result. It contains all the associations that could be extracted. At this point every module and every piece of code appear exactly in one association, no matter in how many of the input products they originally appeared in. This means one can now look up which code pieces implement which modules

by looking at the association that contains the module of interest and the associated code. However, if the association contains more than that one module then it may also contain additional code implementing other modules. As these modules could not be separated, their code could not be separated either. These modules that appear together in one association could not be separated from each other because they never appeared separately in any of the input products. In these cases either an additional input product has to be added from which the necessary information can be extracted or the separation has to be done manually as a post-processing step.

4. EXPERIMENTAL SETTING

An overview of the implemented system[2] is shown in Figure 6. We now describe each of the steps it consists of.

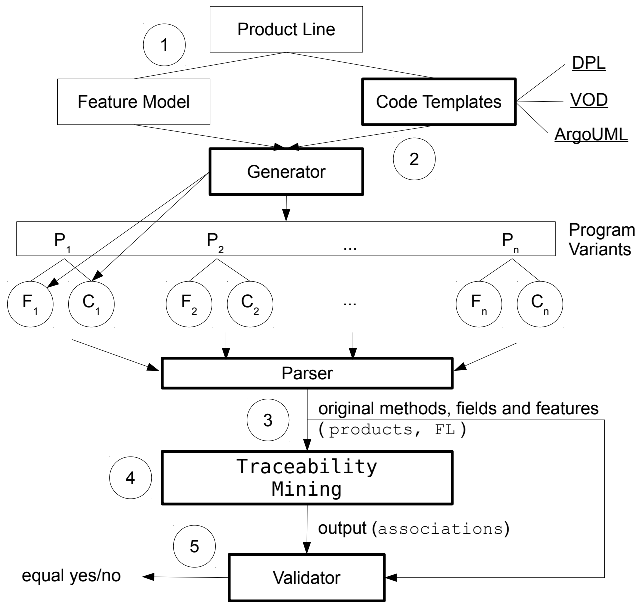


Figure 6: System Overview

Step 1: Product Line and Feature Model. It should be noted that to the best of our knowledge there are no publicly available software repositories from which evolved program variants could be readily mined. Thus, in order to emulate more product variants to evaluate our approach, for some of our case studies we took existing product lines and generated products according to their feature models. The generated products are then used as input for our algorithm to see how well it performs.

Step 2: Templates and Generator. This part of the system sets up the testing environment. We need it to generate products (the Java source files) from a product line so we can use them as input. For that purpose we have the whole source code of a product line in the form of templates where each piece of code is guarded with certain features. The templates are parsed by the code generator using the Apache Velocity template engine [1] or the JavaPp [3], depending on the product line. Code Listing 2 shows an excerpt from the Velocity template file for the class `Canvas`. The lines 2 and 3 are only included in the product if feature `LINE` is selected.

[2]Source Code: `http://www.sea.jku.at/traceability/source`

	VOD	ArgoUML	MM
Mandatory Features	6	3	6 [1]
Optional Features	5	8	8 [1]
Possible Products	32	256	7 [1]
Lines of Code	5.3K+	340K+	5K+
Classes (*.java Files)	42	1915	50
Fields	392	4452	223
Methods	249	16676	422
Unique Code Pieces	642	21128	645
Associations (with Code)	5	26	22
Correctness [%]	100	99.4	99.6
Performance [sec]	0.9	45	1.3
Distinguishability	63.8	7.8	3060.8

[1] Estimated values.

Table 2: Case Studies Data Summary

```
1 #if ($LINE)
2 protected List lines =
3   new LinkedList();
4 #end
```

Code Listing 2: Template snippet of class `Line`

Step 3: Parser. For every given product used as input, the parser reads the features from a text file and extracts all the code elements from the Java source files using the Java Compiler API [5]. The extracted code is on the granularity of class methods and fields.

Step 4: Traceability Mining. This is the core of our approach. The extracted code and features are fed into the algorithm as well as into the validator for later verification. The algorithm computes associations between modules and source code (fields and methods) as output.

Step 5: Validator. The validator receives the original products as well as the output from the algorithm as input. With the traceability information provided by the algorithm the validator reconstructs products with the same features as the original products. The reconstruction is done for each input product separately by taking the features it implements and generating the code elements as the union of the code elements of the output associations corresponding to these features:

$$P' = Reconstruct(P.f, associations)$$

where P' is the reconstructed product and P is the corresponding original product. As final step, each reconstructed product's code elements are compared to the original product's code elements.

5. EVALUATION

An overview of the case studies used for evaluation is shown in Table 2. It proved difficult to get hands on real world software variants that fit our scenario. Therefore for our first two case studies we used existing SPLs from which we generated a number of variants and treated them as if they were independently developed. This is sufficient to

show the correctness of our approach. The third case study however, was taken as is from [4]. In the following we present our evaluation criteria and the results obtained in our three case studies.

5.1 Evaluation Criteria

Based on the experimental setting, we identified three criteria for assessing our algorithm.

Definition 1. *Correctness is the average percentage of code overlap between each original input product and its corresponding reconstructed product (see Section 4 Step 5) using the extracted traceability information.*

$$Correctness = \frac{1}{n} * \sum_{i=1}^{n} \frac{|P_i.c \cap P'_i.c|}{|P_i.c \cup P'_i.c|}$$

where n is the number of products used as input, P_i is an original input product and P'_i is the corresponding reconstructed product.

If all original products are reconstructed in this manner and the comparison shows that they are equal then the extracted traceability information must be correct, at least for the given products. That would mean a 100% value for this metric.

Our algorithm may only err in incorrectly assigning a code element to a module. Consider now that there are two modules and let us assume that our algorithm incorrectly assigns a code element to $\texttt{module}_1$ although it belongs to $\texttt{module}_2$. For any product that includes both modules, this error would remain undetected because together they exhibit the right code elements. However, for any product that contains one of the modules only, the product would either be missing a code element or have an extra code element. Hence, the need to assess correctness by reconstructing and comparing all products used by the algorithm.

Definition 2. *Performance is the execution time of the traceability mining algorithm, not including the experimental setup such as the generation of products or the parsing of the original source code.*

The execution times were measured on an Intel® Core™ i5 Sandy Bridge with 8 GB of memory.

Definition 3. *Distinguishability is the average cardinality of all module sets whose respective associations contain code and at least one module.*

$$Distinguishability = \frac{1}{n} * \sum_{i=1}^{n} |association_i.m|$$

where n is the number of associations that contain code and at least one module and $association_i$ is such an association.

The optimal value for this metric is 1, meaning every association containing code has exactly one module. The measure is important because our approach can only distinguish modules if one of them appears in at least one product in which the other doesn't. Consider, for example, mandatory features that all products must have. As they always appear together and never without each other, the corresponding modules and code cannot be distinguished.

5.2 Case Study: Video On Demand (VOD)

The *Video On Demand (VOD)* product line consists of simple video streaming applications. We generated all possible products and used them for evaluation. The achieved correctness was at 100% with a performance of 0.9 seconds. The distinguishability was at 63.8. The lower bound for the number of modules possible in an association due to mandatory features in this case study is at $2^6 - 1 = 63$.

5.3 Case Study: ArgoUML-SPL

The *ArgoUML-SPL* is the SPL for the UML Modelling Tool ArgoUML [6, 2]. Again all possible products were generated and used for the evaluation. The achieved correctness was at 99.4%. Only 150 code pieces out of 21128 could not be associated with the correct modules, in fact, they were not associated with any module at all, since for these code pieces multiple traces existed (see details in Section 5.5 Analysis). The distinguishability was at 7.8 modules per association containing code with the lower bound for the number of modules in an association at $2^3 - 1 = 7$.

5.4 Case Study: MobileMedia (MM)

The third case study we evaluated has 7 product variants obtained from a system called *MobileMedia (MM)*. In contrast to the previous case studies, each variant corresponds to an evolutionary step of the system development [4]. The features for each product were assigned manually by inferring them from the corresponding paper [7]. The number of features ranges from 6 for the smallest to 14 for the largest product.

With all 7 product variants as input the achieved correctness was at 99.6%. Only one piece of code could not be assigned to a module. Taking a closer look at this piece revealed that it was accidentally left over in one of the products where it was not needed anymore. It was correctly assigned after removing it from this product. So in a way our algorithm pointed us at a mistake in one of the original products which we then corrected.

The distinguishability for this case study was 3060.8. This is due to the very small subset of the possible products for this number of features used as input. Most of the modules are higher order derivatives that don't exist in the form of code anyways (see Section 5.5 Analysis). Removing all modules with an order higher than 1 (no interactions between more than 2 features) led to a distinguishability of 7.14 while having no influence on the correctness.

5.5 Analysis

In our experiment, we found that some pieces of code could not be associated with any module. The reason is that, generally speaking, such pieces of code appear in disjunctive features. For example, product $\texttt{P}_1$ uses piece of code `c` in feature `A`, product $\texttt{P}_2$ uses the same piece of code in feature `B`, while `c` is annotated with a condition to include it if feature `A` OR feature `B` is implemented. Even though the two products do not share any common feature they *do* share a common piece of code. Thus code `c` does not have a unique trace, because it is added by multiple disjunctive features and therefore would have to be traced to multiple modules. And currently our approach is limited to code pieces with unique traces. More details will be available in [11].

Another reason for code not being associated with modules can be mistakes in the input products, for example when

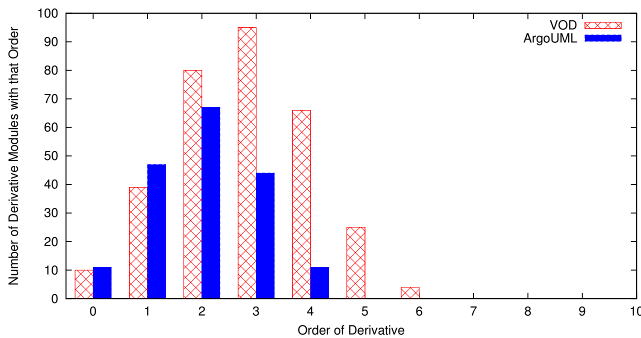


Figure 7: Number of modules per order of derivative for ArgoUML and VOD

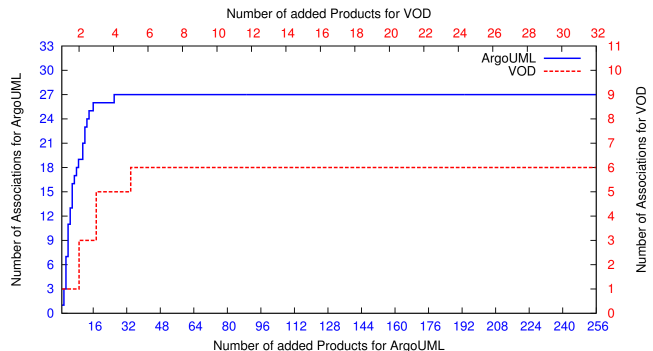


Figure 8: Number of Associations per added Product for ArgoUML and VOD

code was not removed from products where it didn't belong to anymore. In a way, our algorithm points out those mistakes.

In addition to the correct modules, pieces of code were often also associated with a large number of higher order derivatives that, if they existed (in the form of code), would be implemented in this code, which our system cannot know. So basically our algorithm indicates that such code can belong to any (or several) of these modules. In order to avoid that, one could set a threshold for the maximum order that derivatives may have. For example one could assume that in a certain product line, no more than 4 features interact (depending on the coupling). Therefore, derivatives with an order higher than 3 could be omitted. This may drastically improve both the performance as well as the distinguishability and have almost no negative impact on the correctness if the threshold is chosen wisely. It also should be noted, that most of these derivative modules seemed to stem from the undistinguishable mandatory features. Most of the other derivatives could be separated and filtered out (there was no code associated with them). A possibility to avoid that would be to represent all the mandatory features with one single representative feature, as they can't be separated anyway.

In Figure 7 the number of modules with respect to their number of interacting features (which is the order of the derivatives) are shown for VOD and ArgoUML. VOD contains more higher order derivatives than ArgoUML, partially because it has more mandatory features that cannot be distinguished. The highest order of derivatives that would have been possible for both is 10, as both happen to have 11 features. Also, one of the base modules in VOD does not show up because it has no source code at our current code granularity level of methods and fields. Figure 8 shows the number of extracted associations after each additional product that is considered. Only roughly the first 15% of the products provide new associations (this depends on the selected features of the products and in what order they are processed). The remaining products help to get rid of extra derivatives. This shows us, that it would by far not have been necessary to generate all possible product variants and all higher order derivatives, which would have dramatically decreased the runtime and wouldn't be possible in a real world scenario anyway. MobileMedia is not included in the plots because our information about it is not complete as we did not generate the variants ourselves.

6. THREATS TO VALIDITY

In practice only few product variants may be available for input and the automatically extracted traceability information may therefore be rather coarse grain.

The product variations we studied were maintained consistently and thus did not exibit evolutionary differences like bug fixes or feature extensions/changes.

For example a bug fix that was performed on $\texttt{Product}_1$ but not on $\texttt{Product}_2$ would then appear as a feature difference because it is a difference in code. However, this problem only appears in cases where products are no longer maintained (or maintained differently). This is often not the case in practice where it is desired to maintain all variations for as long as they are needed. Obsolete products may no longer be maintained but their traceability may thus also no longer be important.

Additionally, the fact that our approach (for now) does not investigate method bodies shields it from smaller variations and it remains unaffected. Only if bug fixes require larger code changes (including new methods and their calling behavior) we may be affected. However, such larger changes also often go together with feature changes/extensions in which case they can be handled as composite features (extended feature $\texttt{F}_1^{\texttt{ext}}$ is like feature $\texttt{F}_1 + \texttt{F}'_1$).

7. RELATED WORK

For this work we were using only very basic clone detection techniques just above the level of plain string comparison (see clone detection techniques in the work of Roy et al. [13]); we compared code pieces based on their unique Java signatures (e.g. fully qualified class names in combination with method signatures). Comparisons of method bodies on statement level will provide more detailed traceability and is a necessary step towards automatically reverse-engineering a Software Product Line from product variants. We argue that such more advanced clone detection techniques can be utilized in combination with our work presented in this paper to improve the quality and level of detail of the extracted traceability information, without significantly changing our approach.

The work of Siegmund et al. aims to predict non-functional properties of SPL's products by generating and measuring a small set of product variants and approximating each feature's non-functional properties [16]. Our approach works similarly in that we also extract information from a set of

product variants, only that we aim to extract each feature's source code instead of non-functional properties.

Ziadi et al. present a partially automated approach to identify features from source code of product variants in order to help migrate software product variants into a product line [19]. With the same goal in mind, our approach aims to extract traceability information from product variants about which the features are already known.

The work of Rubin et al. aims at generating a product line out of related products [14] by comparing and matching artifacts of these products and merging those that are similar. Their focus is on the formal specification of a product line refactoring operator that puts individual products together into a product line. In [15] they suggest two heuristics for improving the accuracy of feature location techniques when locating distinguishing features. These heuristics are based on information available when looking at multiple product variants together by comparing the code of a variant that contains a feature of interest to one that does not. The features of interest are implemented in the unshared parts of a program which they call a diff set. In the algorithm we presented in this paper we also make use of the information that becomes available when comparing product variants with each other and looking at features and source code they share and don't share.

Xue et al. discuss problems when using information retrieval techniques to identify features and their implementing code when applied to a collection of product variants. To overcome this problems they present an approach to improve feature location in product variants by exploiting commonalities and differences of product variants by software differencing and formal concept analysis so that information retrieval techniques achieve satisfactory results [18].

In the work of Kaestner et al. they examine the impact of the optional feature problem of product lines by means of two case studies and survey different solutions and their trade offs [10]. This problem is also relevant in this paper where we capture dependencies between the implementations of features by means of derivatives, regardless whether these features are independent in the domain or not.

8. CONCLUSIONS AND FUTURE WORK

We introduced an algorithm to extract traceability information between features as well as feature combinations and code in product variants, on the level of class fields and methods. We evaluated our approach with three case studies of different sizes and complexity. More than 99% of the code pieces were correctly assigned in a matter of seconds, even with large products like the ones from the ArgoUML-SPL. The remaining code pieces that could not be assigned to modules fall under our unique trace limitation.

As part of our future work we plan to: i) extend the algorithm to work on granularity below method level, e.g. statements or expressions, including changes in the ordering. This would open the possibility of leveraging advanced clone detection methods, ii) relax or eliminate the unique trace limitation to allow more accurate tracing and handle non-assigned code pieces, iii) enhance the algorithm with static and dynamic analysis of programs, and iv) perform a more detailed evaluation of our algorithm with more case studies as well as only using subsets of all possible product variants.

9. REFERENCES

[1] Apache velocity. http://velocity.apache.org/.

[2] Argouml-spl project. http://argouml-spl.tigris.org/.

[3] Javapp project. http://www.slashdev.ca/javapp/.

[4] Mobilemedia. http://sourceforge.net/projects/mobilemedia/.

[5] Source code analysis using java 6 apis. http://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html.

[6] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In *CSMR*, pages 191–200. IEEE Computer Society, 2011.

[7] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. C. Ferrari, S. S. Khan, F. C. Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, pages 261–270. ACM, 2008.

[8] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. Reverse engineering feature models from programs' feature sets. In *WCRE*, pages 308–312. IEEE Computer Society, 2011.

[9] S. Hutchesson and J. A. McDermid. Towards cost-effective high-assurance software product lines: The need for property-preserving transformations. In *SPLC*, pages 55–64, 2011.

[10] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. S. Batory, and G. Saake. On the impact of the optional feature problem: analysis and case studies. In *SPLC*, pages 181–190, 2009.

[11] L. Linsbauer. Reverse engineering variability from product variants. In *Master's Thesis, Johannes Kepler University Linz*, to appear in 2013.

[12] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proc. of 28th int. conf. on Software engineering*, ICSE '06, pages 112–121, New York, NY, USA, 2006. ACM.

[13] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[14] J. Rubin and M. Chechik. Combining related products into product lines. In *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2012.

[15] J. Rubin and M. Chechik. Locating distinguishing features using diff sets. In *ASE*, pages 242–245. ACM, 2012.

[16] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *SPLC*, pages 160–169, 2011.

[17] F. J. van d. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[18] Y. Xue, Z. Xing, and S. Jarzabek. Feature location in a collection of product variants. In *WCRE*, pages 145–154. IEEE Computer Society, 2012.

[19] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane. Feature identification from the source code of product variants. In *CSMR*, pages 417–422. IEEE, 2012.