# Identifying Variability in Object-Oriented Code Using Model-Based Code Mining*

David Wille[1], Michael Tiede[1], Sandro Schulze[2], Christoph Seidl[1], and Ina Schaefer[1]

[1] TU Braunschweig, Germany
{d.wille, m.tiede, c.seidl, i.schaefer}@tu-brauschweig.de
[2] TU Hamburg-Harburg, Germany
sandro.schulze@tuhh.de

**Abstract.** A large set of object-oriented programming (OOP) languages exists to realize software for different purposes. Companies often create variants of their existing software by copying and modifying them to changed requirements. While these so-called clone-and-own approaches allow to save money in short-term, they expose the company to severe risks regarding long-term evolution and product quality. The main reason is the high manual maintenance effort which is needed due to the unknown relations between variants. In this paper, we introduce a model-based approach to identify variability information for OOP code, allowing companies to better understand and manage variability between their variants. This information allows to improve maintenance of the variants and to transition from single variant development to more elaborate reuse strategies such as software product lines. We demonstrate the applicability of our approach by means of a case study analyzing variants generated from an existing software product line and comparing our findings to the managed reuse strategy.

## 1 Introduction

*Object-oriented programming (OOP)* languages are widely used to develop software for different applications. Depending on the systems' requirements, different languages provide a suitable degree of abstraction from the underlying hardware. For instance, C++ allows development of software with real-time requirements (e.g., in embedded systems), while Python or Java are often used to develop software for desktop applications (e.g., administration tools).

Software companies often are specialized in developing solutions for a certain domain (e.g., logistics software) and their applications have a common functionality. However, customers often have differing requirements and, thus, *variants* of the software need to be developed. For instance, a certain functionality is required by a customer for a company-specific process (e.g., the way stock data

* This work was partially supported by the DFG (German Research Foundation) under grant SCHA1635/2-2 and by the European Commission within the project HyVar (grant agreement H2020-644298).)

is stored). Thus, copying existing variants and modifying them to changed requirements is common practice to reuse functionality and reduce the development time for variants. This so-called *clone-and-own* practice allows companies to save money during the creation of a *software family* consisting of related variants with slight differences. However, companies rarely document the relations of created variants and existing errors are propagated between variants. As a consequence, identifying and fixing errors becomes a time-consuming and costly task because corresponding parts have to be identified by manually comparing the variants. Thus, the practice of clone-and-own is considered harmful to the long-term development process [5, 7, 12].

One solution to overcome clone-and-own related problems is introducing managed reuse of functionality to the created family of product variants instead of maintaining independent variants in isolation. Reverse engineering *variability information* (i.e., common and varying parts) from the related product variants allows developers to identify relations between them and use these insights during maintenance. For instance, changes can be propagated more easily between the variants and managed reuse of existing functionality is possible. Thus, the software quality improves and shorter development times for new products are possible [13]. In previous work, we successfully demonstrated *family mining*, a variability mining technique for block-based modeling languages [18, 19].

In this paper, we introduce a model-based *code mining* framework, which allows to identify the variability between related variants realized in source code of OOP languages. The framework uses a language-independent meta-model as a data structure and executes all comparisons between the analyzed artifacts on this basis. In particular, we make the following contributions:

- We present a language-independent meta-model for OOP languages.
- We introduce a model-based code mining framework allowing to identify the variability between OOP code variants.
- We demonstrate the applicability of our approach by means of a case study.

This paper is structured as follows: Section 2 gives an overview of OOP languages, model-based software development, and family mining. Section 3 describes our model-based code mining approach and the language-independent meta-model for OOP languages. Section 4 provides a case study to show applicability of model-based code mining for OOP code. Section 5 discusses related work and Section 6 concludes with an outlook to future work.

## 2 Background

In this section, we give an overview of OOP languages (cf. Section 2.1), model-based development (cf. Section 2.2), and variability mining (cf. Section 2.3).

### 2.1 Object-Oriented Programming Languages

OOP languages are widely used in different domains to develop software systems for various application scenarios. For this reason, a large number of OOP lan-

```python
class Rectangle:
  def __init__(self, a, b):
    self.a = a
    self.b = b

  def calculateArea(self):
    return self.a * self.b
```

Listing 1.1: PYTHON class

```java
public class Rectangle {
  private int a, b;

  public Rectangle(int a, int b) {
    this.a = a;
    this.b = b;
  }

  public int calculateArea() {
    return a * b;
  }
}
```

Listing 1.2: JAVA class

```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle {
  private:
    int a, b;
  public:
    Rectangle(int a, int b);
    int calculateArea();
};
#endif /* RECTANGLE_H */
```

Listing 1.3: C++ header

```cpp
#include "Rectangle.h"

Rectangle::Rectangle(int a, int b) {
  this->a = a;
  this->b = b;
}

int Rectangle::calculateArea() {
  return a * b;
}
```

Listing 1.4: C++ class

guages exists to realize information structures supporting different requirements (e.g., developing desktop and server software).

In Listings 1.1 to 1.4, we show code examples for PYTHON, JAVA, and C++. All three languages provide similar concepts but use different names for some of them. They consist of *classifiers* (also referred to as *classes*), which are part of a *namespace* to organize and encapsulate them. Classes define the structure for objects containing *variables* to store values with different *types* (e.g., integers or characters) and *methods* to execute operations on them. Each example shows a `Rectangle` class storing two integers `a` and `b` to calculate the rectangle's area using the method `calculateArea()`. Methods have a *signature*, which is defined by their *visibility modifier* (e.g., `public` or `private`), *return type* (i.e., the type of the method's result), method name, and *parameters*. Visibility modifiers are used to control the accessibility of variables and methods inside of classes (i.e., whether other classes can access them). Parameters pass information to *constructors* (i.e., the method initializing objects during object creation) or methods to further process them. Although the basic structure of most OOP languages is very similar, certain differences exist. For example, C++ classes are normally defined in *header* files (cf., Listing 1.3), while their concrete implementation is defined in a separate implementation file (cf. Listing 1.4). Another example is PYTHON as it does not provide explicit visibility modifiers (i.e., all class variables and methods are accessible by other classes).

## 2.2 Model-Based Software Development

Model-based software development uses *meta-models* to define the language elements that can be used for a specific language. *Models* as concrete instances of

meta-models allow to store data with regard to the provided language elements. With a suitable parser, source code can be perceived as a model conforming with a specific meta-model. The *concrete syntax* of a language is defined by a *grammar* and allows parser technology to create such a model representation from source code by transforming it to an instance of the meta-model. Developers use the concrete syntax (e.g., the textual syntax of JAVA source code) provided by many languages to implement their software (i.e., models). Meta-models are also referred to as a language's *abstract syntax* because they define how model instances have to be realized in order to conform with the language.

### 2.3 Family Mining

Companies often develop software for a specific domain (e.g., the logistics domain) with a large number of customers. These customers mostly use the same functionality but often have requirements to support certain company-specific tasks (e.g., a tax calculation module for a specific country). As a result, software companies need to develop *variants* of their products. A common strategy to create new variants is cloning-and-owning, which allows to reuse existing functionality, but also introduces risks to the development process (e.g., duplicate bug fixing of errors that were propagated between variants) [5].

A possible solution is to introduce managed reuse of existing functionality to the product family. Software product lines (SPLs) provide a suitable reuse mechanism as they manage all implementation artifacts (e.g., source code) for multiple variants in a single location. *Features* represent configuration options for such SPLs [13] and allow to configure and generate variants from the product family. In previous work, we successfully introduced *family mining*, a variability mining technique, to reverse engineer the *variability information* (i.e., common and varying parts) for block-based languages by using a metric-based compare algorithm [18, 19]. After identifying relations between the compared variants, we merge all analyzed variants in a *150% model* annotating all implementation artifacts with their variability. This information allows to analyze relations between variants and, thus, to improve the maintenance (e.g., by applying changes to all variants more easily). Furthermore, the variability information allows to gradually introduce SPL reuse strategies by mapping artifacts to features [1]. In this paper, we introduce family mining for source code of OOP languages.

## 3 Model-Based Code Mining Framework

In this section, we introduce our model-based code mining framework allowing to identify variability information for related OOP code variants. In Figure 1, we show the corresponding workflow. Our framework expects source code in a supported OOP language (i.e., currently JAVA and C++), which is transferred to an instance of our meta-model (cf. Section 3.1). For our comparisons, we utilize a metric (cf. Section 3.2) to calculate the similarity of compared elements. The identified variability can be exported in form of a summarizing report or a 150%

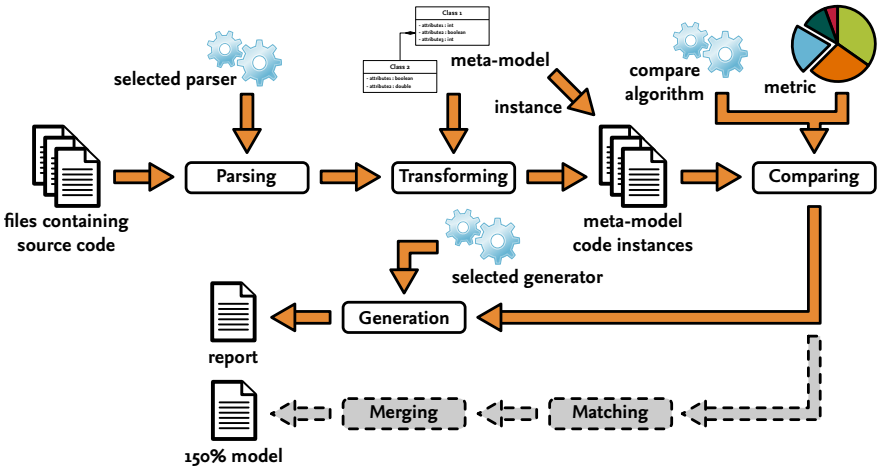


Fig. 1: Workflow for the model-based code mining framework.

model storing all implementation artifacts from the analyzed variants together with their annotated variability.

The first step during variability analysis is to parse the selected OOP code variants using a corresponding parser and to transfer the resulting information to an instance of our meta-model (cf. Section 3.3). Afterwards, we utilize the defined metric to compare the models of the different variants (cf. Section 3.4). The identified variability information can be exported as a summarizing report (cf. Section 3.5). In addition, we discuss the creation of an annotated 150% model but leave a concrete concept and its realization for future work (cf. Section 3.6).

### 3.1 Language-Independent Meta-Model

Using a model representation to identify variability between related code variants allows to abstract from the language-specific textual syntax of the analyzed programming language. This is possible because of the underlying abstract syntax of all programming languages captured in the meta-model. In theory, it would be possible to reuse an existing meta-model (e.g., used to define the language's abstract syntax) or to build one meta-model per analyzed language. However, as a result, our compare algorithms would have to be implemented for each individual meta-model. Instead, we created a language-independent meta-model to have a common data-structure for OOP languages and, thus, can use the same implementation of our algorithms for a large set of these languages (e.g., Java or C++). Such a generic meta-model is possible due to the common language elements of all OOP languages (e.g., classes, methods, and parameters). For complete comparison of all language concepts, we also allow extension of the meta-model with nested types, e.g., to model generics for Java or templates for C++. Besides allowing analysis of code in different OOP languages (e.g., Java or C++) on a common meta-model structure, our language-independent meta-model facilitates *cross-language* code comparisons (e.g., between Java and C++).

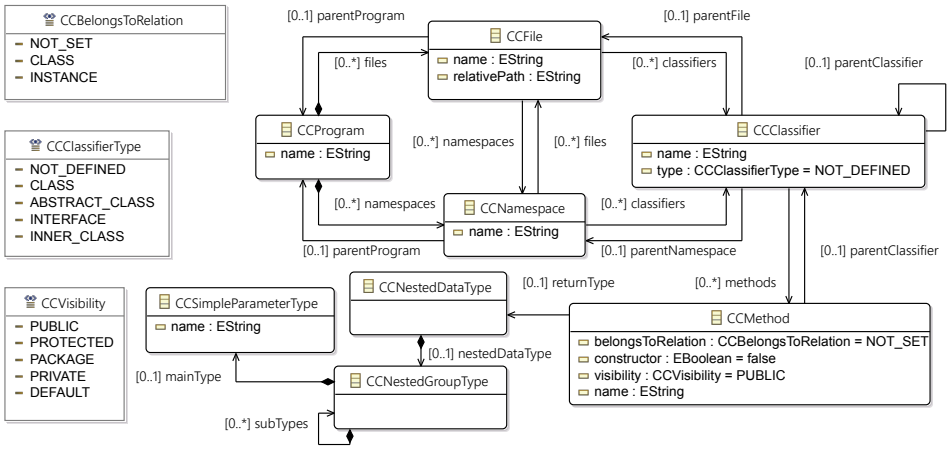


Fig. 2: Excerpt from the language-independent meta-model.

In Figure 2, we show an excerpt from our language-independent meta-model. As we can see, *programs* contain a set of *files* and *namespaces*. Files are contained in a namespace and contain *classifiers* with different types (e.g., abstract classes or interfaces). Classifiers can contain further classifiers (i.e., inner classes) and define a set of methods. Methods can be declared as constructors and have a name, visibility, and *belongs-to-relation*. This relation allows to distinguish between *class methods* (also called *static methods*) and *instance methods*. These methods have a return type, which is represented as a nested data structure allowing to model more complex nested types for different languages. This nested type is used for all data types in the meta-model (e.g., method parameters).

### 3.2 Metric Definition

Using the specified meta-model, we define a metric for the comparison of code elements from OOP programs. This metric allows to assign different importance to the properties of compared elements and, thus, to influence their impact on the elements' similarity. The overall sum of all weights for an element (e.g., methods or fields) is normalized to be in the interval [0..1] to make calculated similarity values comparable. In Table 1, we present the properties considered during the comparison of methods and fields together with the corresponding weights, which have empirically proven to be sensible values and have been used for our case study (cf. Section 4). During the comparison of methods, we assign the highest weight to their names because developers select these names for specific reasons and, thus, these properties are an important indicator for similarity. We are not just interested in equality but also in cases, which are related variants with reduced similarity (i.e., we allow minor deviations in the names). The return type highly influences the behavior of corresponding methods as it determines the expected result. However, multiple methods can exist with the same return type and, thus, we assign a lower weight to them. During the comparison of method parameters, we assign a lower weight to their types than to

Table 1: Metric used for the variability mining of OOP code

| **Method Weights** | | **Field Weights** | |
|---|---|---|---|
| *Property* | *Weight* | *Property* | *Weight* |
| Overall Modifier | 0.05 | Name | 0.4 |
| ◦ *Visibility* | *0.25* | Data Type | 0.4 |
| ◦ *Constant* | *0.25* | Overall Modifier | 0.2 |
| ◦ *Static* | *0.25* | ◦ *Visibility* | *0.33* |
| ◦ *Abstract* | *0.25* | ◦ *Constant* | *0.33* |
| Return Type | 0.2 | ◦ *Belongs-to-Relation* | *0.33* |
| Name | 0.45 | | |
| Parameter Names | 0.2 | | |
| Parameter Types | 0.1 | | |

corresponding names. Here, multiple methods with the same parameter types might exist and only their names make the parameters distinguishable. Finally, the modifiers of methods are weighted with the lowest value as they have relatively little impact on the similarity. The weight for modifiers is subdivided into four weights for the corresponding visibility, constant (i.e., the method cannot change class fields), static (i.e., the method can be invoked without creating a class object), and abstract modifiers (i.e., the method has to be overridden by a concrete implementation). In addition to the methods' signatures, we also consider their body during comparison. For the overall similarity of two methods, we empirically identified that weighting the signature similarity with 0.6 and the body similarity with 0.4 allows to calculate sensible results. During comparison of fields, we consider names, data types and modifiers. Here, we assigned a low weight to the modifiers as they only slightly influence the behavior of fields. The weight for field modifiers is subdivided into weights for the visibility, constant (i.e., the field cannot be changed), and belongs-to-relation modifiers.

For configuring the overall mining process and particularly the metric, we allow developers to define configuration values to specify programs that should be compared, to select the used compare algorithms, and to configure the metric's weights. Here, we allow to define relations and corresponding similarity weights for similar types, which are not 100% equal but still can be regarded as similar types (e.g., lists and arrays). This way, we allow adaptability and custom-tailored code mining when applying our algorithms to programs with different settings.

### 3.3 Parsing and Transforming Source Code Files

The first step is to parse the source code files that should be analyzed using a suitable parser. Our current implementation supports the import of Java and C++ source code. For Java source code we use the Java Model Parser and Printer (JaMoPP)[3], which parses provided Java code to the JaMoPP meta-model. The resulting meta-model instances are transformed to our language-

[3] `http://www.jamopp.org/`

independent meta-model in order to make the parsed information available for our variability mining algorithms or analysis by users. Our framework allows users to implement and select further parsers to parse and analyze files from other languages. For instance, we used the SCRML[4] parser to parse C++ code. The resulting SRCML XML output of the parsed code is transformed to an instance of our language-independent meta-model and afterwards can be used as input for our code mining algorithms.

### 3.4 Comparing the Source Code Variants

After parsing the source code variants and storing the parsed information in an instance of our language-independent meta-model, we now compare them using our algorithms. Besides utilizing our existing metric and compare algorithm, our framework allows users to realize their own custom-tailored algorithms implementing own metrics. We execute a pairwise comparison for the selected program variants. Our algorithm uses the first program as a basis and compares all other programs with this variant. For all comparisons between two programs, we currently use an *n:m* algorithm. For instance, when comparing the fields of two classes `Class1` and `Class2`, we compare all combinations of the fields.

First, we compare the file names and namespaces of the classes contained in the analyzed programs to identify whether they are contained in the same package and have the same name. Each comparison between two elements is stored in the overall *result model* together with the corresponding similarity value, which is calculated according to the weights from the selected metric. The *result model* allows us, e.g., to analyze and further process the identified relations after the comparisons. Next, we compare the classes contained in the files of analyzed programs. We start class comparisons by comparing their names and the contained fields. Next, we compare the methods' signatures and bodies. Currently, we compare the method bodies by regarding each line of the body as a single string. By comparing these strings, we identify the number of coinciding lines and can calculate the bodies' overall similarity.

For the comparison of names, we calculate the strings' similarity by identifying their edit distance using the *Levenshtein algorithm* [9]. For methods, we split the name into groups starting with upper case characters (camel case notation) and identify the number of equal groups. As OOP languages use differing naming conventions, we allow configuring the used string compare algorithms.

Currently, we regard all elements with a similarity greater or equal to 85% as *mandatory* (i.e., they are regarded as identical). All elements with a similarity below 60% are regarded as *non-related* making each individual *optional* for the software family as they are not similar enough. Elements with a similarity between the mandatory threshold and the optional threshold (i.e., $\geq 60\% \wedge < 85\%$) are regarded as *alternative*, because they still have a high degree of similarity but differ in minor parts (e.g., a method variant has an additional parameter).

[4] `http://www.srcml.org/`

### 3.5 Generating a Report about the Variability Information

After comparing the selected source code variants, we can export the resulting information. Currently, we provide the possibility to store the identified variability in a JSON format or in form of an HTML report. Both formats contain all executed comparisons and, thus, do not contain distinct matchings between elements (i.e., *1:1* relations). The JSON format allows users to further process and analyze the information with other programs. The HTML report is realized with technologies for dynamic websites, allowing users to interactively analyze the information (i.e., the employed metric and the results) and to make annotations. Most importantly, the report allows users to remove relations between elements, which enables them to narrow down the possible relations until distinct *1:1* relations are identified. In Figure 3, we present an excerpt from a report generated during our case study (cf. Section 4). Here, two variants of the method `addEntityType__wrappee__Fly` with differing bodies were compared.

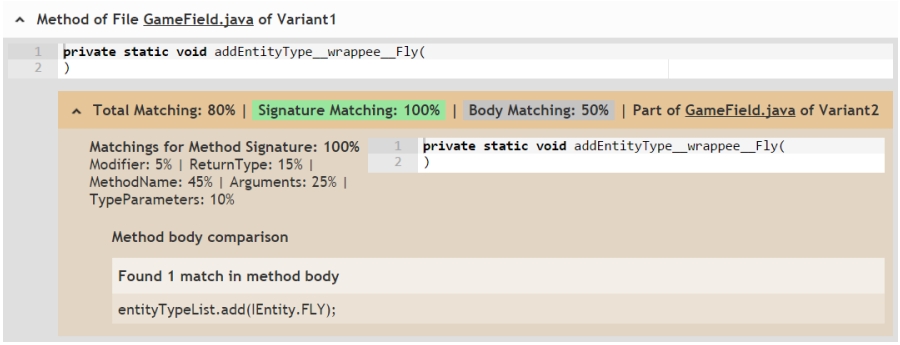


Fig. 3: Excerpt from an example report output.

### 3.6 Creating 150% Model using the Variability Information

In addition to the report generation, we present another export possibility. Using the identified relations, it is possible to create an SPL in the form of a 150% model containing all implementation artifacts from the compared variants annotated with explicit variability information and their containing variants. We realized such a solution in previous work for family mining of block-based languages [18, 19], thus, it seems sensible for code as well. However, as we currently did not realize this option, we marked the corresponding path in Figure 1 with dashed arrows and plan to realize the 150% model generation in future work. To merge the variability information into a 150% model, we first need to identify distinct matches between the different elements from the compared source codes. Thus, we need to transform the *n:m* comparison results into *1:1* relations, allowing to identify unique variability between the corresponding elements. The resulting list of distinct matches can be used to merge a 150% model by adding all compared elements to a single file and annotating their variability (i.e., whether they are part of all variants or only contained in particular variants). In previous work,

for block-based models [18, 19], we already implemented semi-automatic algorithms for distinctively matching model elements, which can easily be adapted for our use case. These algorithms identify all comparisons containing a certain element and select the comparison with the highest similarity. After merging the variability information, the resulting 150% model can be used to analyze the variability of a software family and to generate all contained variants.

## 4 Case Study

We applied the presented variability mining technique in a case study to source code variants generated from an SPL case study. During this case study, we evaluated the following research questions to analyze the approach's feasibility.

**RQ1 – Language Independence:** *Does the proposed language-independent meta-model actually support storing code from different OOP languages?*

**RQ2 – Correctness:** *Is the proposed variability mining technique capable of correctly identifying variability information between related code variants with regard to the variability modeled in the used SPL case study?*

### 4.1 Setup & Methodology

The SnakeFOP case study is part of the FeatureIDE [17] example library and was realized by decomposing an existing Java implementation of the game Snake with 28 classes, 197 methods, and 133 fields into 21 features [8]. Using FeatureIDE, we generated all 5580 possible variants [8] from the SnakeFOP implementation. Although the SnakeFOP case study was not realized using clone-and-own techniques, it allows to evaluate whether our code variability mining approach is able to identify variability between variants correctly by providing a ground truth. For our evaluation, we selected 30 variants subdivided into three subsets each containing 10 variants. We selected the variants by sorting them into three categories (Small, Medium, and Large) according to their average number of *lines of code (LOC)*. In Table 2, we show the three categories together with information about the average LOC per variant, the average number of executed comparisons, the average number of selected features, and the average runtime of the comparisons. All comparisons were executed by a single developer using HTML report generation and the metric weights from Table 1 in Section 3.2 on a laptop with a 2.7 GHz Intel i7 processor and 12 GB RAM. For each category, the first variant was compared with all remaining variants (i.e, in total 27 comparisons were executed) and each comparison between two variants was executed 10 times to reduce the influence of inaccurate runtime measuring.

In addition, to demonstrate the language-independence of our meta-model, we used a set of C++ examples with a variety of language features (i.e., classes, methods, and fields). Executing a preliminary evaluation, we used these examples to analyze whether we can transform them to our meta-model and execute our variability analysis on the resulting meta-model instances.

Table 2: Details of the selected variants for the case study.

| Size | Avg. LOC | Avg. Comparisons | Avg. Features | Avg. Runtime |
|---|---|---|---|---|
| Small | 2037.3 | 33189.8 | 8.7 | 25457,0 ms |
| Medium | 2124.3 | 48150.0 | 13.6 | 30301,2 ms |
| Large | 2274.0 | 54744.0 | 16.5 | 45278,1 ms |

### 4.2 Results & Discussion

Next, we report our results and discuss them to answer our research questions.

*RQ1 – Language Independence:* During the preliminary analysis of the meta-model's language independence, we successfully transformed the used Java variants and the C++ examples to our meta-model. Using the discussed compare algorithms, we were able to generate sensible results conforming with expectations of two consulted developers with experience in SPL design. Although, we only analyzed the meta-model's capabilities with two languages, we are confident that the meta-model is capable of storing code for different OOP languages because of their common structure.

*RQ2 – Correctness:* After executing the comparisons for the selected categories, we manually compared the identified possible relations from the generated HTML reports with the ground truth (i.e., the SnakeFOP SPL). During this analysis, we found that our algorithms were able to identify all variability relations correctly, which can be accounted for by the used *n:m* algorithms as they generate each possible combination. However, we used the dynamic functionality of the HTML reports to manually reduce the relations to distinct combinations (i.e., each element from one variant is matched to exactly one element from the other variant) and assessed the corresponding similarity values. After consulting two developers with experience in SPL design, we identified that about 70% of the values were reasonable. Here, we identified that the calculated field similarities were sensible and only the similarity for methods did not always meet the expectations. While, we classified about 90% of the method signature similarities as sensible, only 50% of the method body similarities met the expectations. Main reasons were methods of different size where the statements from the smaller method were all contained in the larger method but the overall similarity was calculated in accordance to the larger method.

### 4.3 Threats to Validity

During the case study, a single developer manually evaluated the results of our algorithms and compared them to the ground truth (i.e., the variability information from the SnakeFOP SPL). The intuition of other researchers or developers might differ. However, the evaluating developer consulted two developers with experience in SPL design during the classification of the results and, thus, we are confident that the results are close to the expectations by other developers. In addition, our approach uses metrics as heuristic weights to calculate the

similarity of compared elements. These metrics are highly dependent on human intuition and the created results might not always conform with expectations of other developers. However, we already gained experience in identifying variability during our research on block-based languages [18, 19, 20] and, thus, we argue that the employed metrics create results close to the domain experts' intuition.

Our case study is limited to a single scenario from an SPL with JAVA as OOP language and a preliminary analysis for C++ code. Thus, we cannot generalize the applicability of our algorithms to all OOP languages. However, we are confident that our ideas are transferable to further languages as the JAVA case study and the analysis of C++ code variants showed promising results. In addition, we realized our algorithms by iteratively adding functionality to support comparison of different OOP language features. Only afterwards, we executed comparisons using the selected variants. Thus, we did not overfit our implementation for the used case study and are confident that the algorithms are capable of handling other code. Furthermore, due to the extensible design of our framework and the common elements of all OOP languages considered during the design of our meta-model, we are confident that our algorithms are able to handle further OOP languages (e.g., PYTHON).

## 5 Related Work

Several techniques have been proposed that aim at revealing the similarity of source code. We contrast this work to our technique proposed in this paper.

*Clone Detection:* Clone detection is a prominent technique that has been proposed to detect similarities in source code within or between software systems [14]. As such, it is paramount to cope with implementations that result from clone-and-own. While many techniques exist that mainly differ in the underlying program representation (e.g., text-based, token-based, or tree-based), all of them focus on detecting similarities between code fragments. For example, *Hemel et al.* analyze the result of clone detection to perform a large-scale variability analysis with the goal to detect non-propagated patches across LINUX distributions [4]. In contrast to our work, these techniques only focus on cloned parts (except for defining different clone types) and, most notably, do not consider variability semantics, such as relations between similar code fragments.

*Code Merging:* Using identified variability information, approaches exist to merge software variants in different contexts. *Yoshimura et al.* merge several individual systems in a software family by using a clone classification and analysis approach [21]. However, since their focus is on migration, they do not define variation points (e.g., optional or alternative parts) but only assess the technical opportunities of merging parts of the analyzed systems. *Rubin et al.* propose an advanced algorithm with n-way comparison, which tackles the combinatorial explosion when comparing multiple software systems [16]. However, we provide variability information that could be used for merging in a migration process, while Rubin et al. only focus on merging, thus, omitting variability information.

*Reverse Engineering:* In addition to the mentioned approaches, there exists work to explicitly reverse engineer variability information. *Klatt et al.* use program dependency graphs (PDGs) and difference analysis to recover variability information between variants of features [6]. Their approach aims at reducing the manual effort of merging these features and focuses on corresponding differences in the code. In contrast, we focus on identifying variability between entire variants. While their approach exploits a language-dependent PDG, we use a flexible meta-model allowing to analyze different languages. *Martinez et al.* propose a generic framework that aims at identifying variability across multiple artifacts (e.g., models, code, and documentation) [11]. To this end, they provide a generic feature-oriented process, where features are explicitly identified, analyzed, and located. Based on this information, a concept for migration to an SPL is proposed. In contrast, we solely focus on source code and do not use feature information while still revealing variability across variants. Moreover, we do not strictly couple our approach with SPL migration (although it could be used in such a process) but also support other use cases such as change propagation. *Linsbauer et al.* provide an approach for extracting variation points between variants and the corresponding feature-to-code mapping [10]. While they also work on implementation artifacts, they require upfront knowledge about features implemented in each variant. In contrast, we do not include feature information and just relate code elements to each other by means of variability relations. *Duszynski et al.* transform variants to a more abstract representation of the system and identify commonalities of variants by building unions of occurrence matrices [2]. While their approach is limited to comparisons of these matrices, we also exploit more syntactical information of the underlying languages. Similar to Duszynski et al., we are able to apply our approach to different languages because our meta-model abstracts from too specific language elements. However, our approach provides a more detailed summary of the results on statement level, complemented by an overall assessment of similarity on block level (e.g. for methods).

*Rubin et al.* propose a framework for refactoring products into a more feature-centric SPL engineering process [15]. Similar to us, their approach encompasses dedicated phases for comparing and matching. However, it is only applicable to UML and EMF models and mainly focuses on formalizing the merge operator. In contrast, we focus on explicating variability between variants for maintenance activities. *Frenzel et al.* propose an approach to compare product variants on the architectural level [3]. Compared to our approach, their work differs in the considered artifacts (architectural models instead of source code), the general technique (clone detection instead of similarity analysis with variability semantics), and the objective (migration instead of maintenance).

## 6 Conclusion and Future Work

In this paper, we introduced a model-based variability mining technique for OOP programming languages. Our approach parses a set of source code variants to a language-independent meta-model and executes comparisons between the

variants based on this representation. Currently, we realized an *n:m* algorithm to identify the corresponding variability and to generate summarizing reports.

In future work, we plan to optimize our algorithm by automatically exploring the parsed models and ruling out improbable relations to reduce the overall number of comparisons. Furthermore, we plan to adapt algorithms from previous work [18, 19] to identify distinct matches from the comparisons and merge all implementation artifacts into a 150% model together with their annotated variability information. Using the resulting 150% models, we plan to realize seamless transition from clone-and-own based variant creation to software product lines, for example, by generating annotated code (e.g., using C preprocessors) and corresponding feature mappings. Furthermore, we plan to improve the similarity calculation for method bodies by analyzing them on a more fine grained level.

## References


[1] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer. "Flexible Product Line Engineering with a Virtual Platform". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ICSE '14. ACM, 2014, pp. 532–535.

[2] S. Duszynski, J. Knodel, and M. Becker. "Analyzing the Source Code of Multiple Software Variants for Reuse Potential". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. WCRE '11. IEEE, 2011, pp. 303–307.

[3] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann. "Extending the Reflexion Method For Consolidating Software Variants Into Product Lines". In: *Software Quality Journal* 17.4 (2009), pp. 331–366.

[4] A. Hemel and R. Koschke. "Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. WCRE '12. IEEE, 2012, pp. 357–366.

[5] C. Kapser and M. W. Godfrey. ""Cloning Considered Harmful" Considered Harmful". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. WCRE '06. IEEE, 2006, pp. 19–28.

[6] B. Klatt, K. Krogmann, and C. Seidl. "Program Dependency Analysis for Consolidating Customized Product Copies". In: *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)*. ICSME '14. IEEE, 2014, pp. 496–500.

[7] R. Koschke. "Survey of Research on Software Clones". In: *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[8] S. Krieter, R. Schröter, W. Fenske, and G. Saake. "Use-Case-Specific Source-Code Documentation for Feature-Oriented Programming". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. VaMoS '15. ACM, 2015, 27:27–27:34.

[9] V. I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals”. In: *Soviet physics doklady*. Vol. 10. 8. 1966.

[10] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. “Variability extraction and modeling for product variants”. In: *Software & Systems Modeling* (2016), pp. 1–21.

[11] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon. “Bottom-up adoption of software product lines: a generic and extensible approach”. In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. SPLC ’15. ACM, 2015, pp. 101–110.

[12] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. “Software Quality Analysis by Code Clones in Industrial Legacy Software”. In: *Proc. of the Intl. Symposium on Software Metrics (METRICS)*. METRICS ’02. IEEE, 2002, pp. 87–94.

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

[14] C. K. Roy, J. R. Cordy, and R. Koschke. “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach”. In: *Science of Computer Programming* 74.7 (2009), pp. 470–495.

[15] J. Rubin and M. Chechik. “Combining Related Products into Product Lines”. In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 7212. Lecture Notes in Computer Science. Springer, 2012, pp. 285–300.

[16] J. Rubin and M. Chechik. “N-way Model Merging”. In: *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ESEC/FSE ’13. ACM, 2013, pp. 301–311.

[17] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. “FeatureIDE: An Extensible Framework for Feature-oriented Software Development”. In: *Science of Computer Programming* 79 (2014), pp. 70–85.

[18] D. Wille. “Managing Lots of Models: The FaMine Approach”. In: *Proc. of the Intl. Symposium Foundations of Software Engineering (FSE)*. FSE ’14. ACM, 2014, pp. 817–819.

[19] D. Wille, S. Schulze, C. Seidl, and I. Schaefer. “Custom-Tailored Variability Mining for Block-Based Languages”. In: *Proc. of the Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER)*. SANER ’16. IEEE, 2016.

[20] D. Wille, S. Holthusen, S. Schulze, and I. Schaefer. “Interface Variability in Family Model Mining”. In: *Proc. of the Intl. Workshop on Model-Driven Approaches in Software Product Line Engineering (MAPLE)*. SPLC ’13. ACM, 2013, pp. 44–51.

[21] K. Yoshimura, D. Ganesan, and D. Muthig. “Assessing merge potential of existing engine control systems into a product line”. In: *Proc. of the Intl. Workshop on Software Engineering for Automotive Systems (SEAS)*. SEAS ’06. ACM, 2006, pp. 61–67.