Mapping Features to Automatically Identified Object-Oriented Variability Implementations

The case of ArgoUML-SPL

Johann Mortara
johann.mortara@univ-cotedazur.fr
Université Côte d'Azur, CNRS, I3S, France

Xhevahire Tërnava
xhevahire.ternava@lip6.fr
Sorbonne Université, LIP6, Paris, France

Philippe Collet
philippe.collet@univ-cotedazur.fr
Université Côte d'Azur, CNRS, I3S, France

ABSTRACT

In Software Product Line (SPL) engineering, mapping domain features to existing code assets is essential for variability management. When variability is already implemented through Object-Oriented (OO) techniques, it is too costly and error-prone to refactor assets in terms of features or to use feature annotations. In this work, we delve into the possible usage of automatically identified variation points with variants in an OO code base to enable feature mapping from the domain level. We report on an experiment conducted over ArgoUML-SPL, using its code as input for automatic detection through the *symfinder* toolchain, and the previously devised domain features as a ground truth. We analyse the relevance of the identified variation points with variants *w.r.t.* domain features, adapting *precision* and *recall* measures. This shows that the approach is feasible, that an automatic mapping can be envisaged, and also that the *symfinder* visualization is adapted to this process with some slight additions.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Variability traceability, automatic identification of variation points, understanding software variability, software product lines

1 INTRODUCTION

SPL engineering aims at providing a framework to manage variability within a family of software products [6, 15]. By selecting domain features, multiple code assets corresponding to these features are assembled to derive a custom product. A proper and consistent mapping between domain features and the corresponding variability implementation places in code assets is of prime importance for the architecture of any SPL [2, 5]. Within the SPL paradigm, variabilities at the domain and implementation levels are captured separately [6, 15]. Commonly a feature model is used at the domain level whereas many different implementation techniques can be found at the code level, from preprocessor directives, to feature modules, specific annotations, or more classic object-oriented (OO) techniques and patterns [27].

However many variability-rich software systems do not follow a complete SPL approach. It is then needed to extract or refactor the existing implementations to map the domain features to the appropriate places in the code assets. When it is done through feature modules [3] or when conditional compilation directives are used [12, 19], the mapping is facilitated, but currently, no technique supports the identification and mapping of OO variability implementation techniques to domain features. In our previous work [30], we defined an approach to identify the implementation of different kinds of variation points (*vp*-s) and variants [1] by relying on the concept of symmetry in OO structures [32, 33].

We also provided a tooled support with *symfinder*, a toolchain that automates this identification in a single Java code-base and generates a web-based visualization [24]. While it has been successfully applied to several realistic systems, it has not been explored yet whether the automatically identified *vp*-s with variants are relevant for the mapping of domain features to code assets, and how to facilitate this process. In this context, the mapping is not trivial at all, as different kinds of *vp*-s are to be considered, and the mapping itself can be N to M, namely a feature can be implemented by several *vp*-s and a *vp* can be mapped to several domain features. In order to validate the relevance of our automatic identification approach by *symfinder*, we need to evaluate whether a mapping between domain features and our identified *vp*-s of a targeted system is feasible, being it manual for now. Then, to what degree it can be automated.

In this paper, we thus explore the possibility of using the *vp*-s with variants automatically identified by the *symfinder* toolchain as input for a manual mapping process to domain features that preexist. To do so, we study the case of ArgoUML-SPL [8], an SPL version of ArgoUML. ArgoUML is an editor for UML diagrams, which is implemented in Java and thus uses extensively OO mechanisms. The ArgoUML-SPL [8] itself is an SPL created on top of its code by using annotations for conditional compilation to directly delimit feature code. This SPL has already been used by the community as a case study and we take for our study its ground truth [22], which contains a set of trace links for its domain features obtained by analysing these annotations (Section 2). We run *symfinder* on the source code of the ArgoUML-SPL to automatically extract *vp*-s with variants. In order to evaluate the relevance of the symmetry-based automatic identification we adapt the *precision* and *recall* measures and analyse the results of a manual mapping on the case study (Section 3). We then discuss observations on feature tangling and scattering, as well as on feature naming to envisage a possible automation of the mapping. We also discuss some improvements made on the *symfinder* visualization while analysing the case study (Section 4). Finally, we discuss related work (Section 5) and conclude the paper by evoking future work (Section 6).

[1] Their definition is given in Section 2.2

2 BACKGROUND

In the following is introduced the ArgoUML-SPL case study with its ground truth that we use to conduct the feature mapping experiment. Then, we present the *symfinder* toolchain and the obtained results of its application on the ArgoUML-SPL's code assets.

2.1 ArgoUML-SPL ground truth

ArgoUML [2] is an open source UML modeling tool implemented in Java language. In the software product line community it is used as a realistic case study for demonstrating the basic challenges for refactoring a single code base system with variability into an SPL [8]. The extracted ArgoUML-SPL, with its ground truth, was also recently proposed [22] and used [9, 23] as a benchmark for evaluating the feature location techniques. The considered ArgoUML-SPL ground truth [22] consists of a feature model (FM) and traces of its optional features to the annotated reusable code assets.

The extracted ArgoUML-SPL uses a form of conditional compilations, based on the javapp preprocessor, to modularize and delimit a given feature's code [8]. It consists of 11 features, given in the feature model (FM) in Figure 1. The abstract feature `ArgoUML-SPL` represents conceptually the SPL domain, which has 2 mandatory features, `Diagrams` and `Class`, and 8 optional features, `State`, `Activity`, `Use Case`, `Collaboration`, `Deployment`, `Sequence`, `Cognitive Support`, and `Logging`.

The original, non-SPL based, ArgoUML [3] consists of 120,348 LoC. The 8 extracted optional features with their combinations and negations are traced to 31% of its LoC, whereas the other 69% are the core assets of the extracted SPL. These core assets contain also other features that are not extracted, such as the features for rendering diagrams in the screen, for persistence, internationalization, code generation, or reverse engineering [8].

The ArgoUML-SPL benchmark provides a ground truth of traces for the 8 optional features to their respective conditional compilations in code assets. These traces appear as links of a given feature to a complete class, class refinement (*i.e.*, referencing statements within a single class), complete method, or method refinement (*i.e.*, referencing statements within a single method) [22][4]. The benchmark distinguishes the traces of the 8 individual optional features, their 14 feature combinations, with two and three features, and 2 negated individual features. In Table 1 is given a summary of the ArgoUML-SPL ground truth for the 24 features, including their combinations and negations, with their respective number of classes where they are traced to. These traces have a normalized granularity, as explained in Section 3.2. For simplification, feature combinations and negations in Table 1 are presented by numbers, showing which optional features are combined or negated. For instance, the entry $[2-8]$ indicates that there is a feature combination between [2] `Activity` and [8] `Logging`. This `Activity_and_Logging` feature combination has traces to 3 classes in code assets. Similarly, for feature negation, the $\neg[8]$ entry indicates that 4 classes in code assets are needed when feature [8] `Logging` is not present.

[2]https://www.openhub.net/p/argouml
[3]https://github.com/argouml-tigris-org/argouml/tree/master/src/argouml-app
[4]While the term *trace links* is used in the ground truth, we will distinguish from this term in our mapping experiment by using *mapping links* for *vp*-s and variants mapped to features, although both of them have the same meaning.

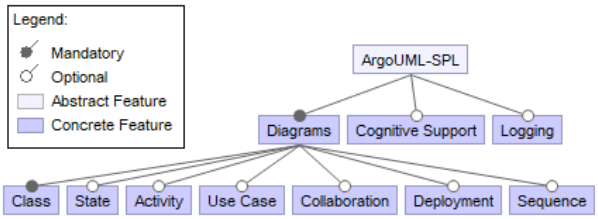

Figure 1: Feature model of ArgoUML-SPL, adapted from [8]

Table 1: ArgoUML-SPL ground truth [22]. Eight optional features, their fourteen feature combinations, and two single feature negations, with their respective number of traces

ID	Feature	#traces	ID	#traces	ID	#traces
[1]	State	98	$[2-8]$	3	$[5-8]$	4
[2]	Activity	94	$[2-1]$	7	$[5-3]$	1
[3]	Use Case	52	$[7-5]$	15	$[8-6]$	16
[4]	Collaboration	48	$[7-8]$	15	$[8-1]$	4
[5]	Deployment	32	$[7-6]$	3	$[8-3]$	4
[6]	Sequence	79	$[4-8]$	3	$[6-1]$	1
[7]	Cognitive	236	$[4-8-6]$	2	$\neg[7]$	1
[8]	Logging	214	$[4-6]$	5	$\neg[8]$	4

```
{
  "nodes": [
    {
      "types": [
        "CLASS", "ABSTRACT", "STRATEGY", "DECORATOR",
        "VP", "METHOD_LEVEL_VP", "VARIANT"
      ],
      "constructorVPs": 1,
      "methodVariants": 5,
      "classVariants": 18,
      "methodVPs": 2,
      "constructorVariants": 3,
      "name": "org.argouml.uml.diagram.ui.FigNodeModelElement"
    },...
  ],
  "links": [
    {
      "type": "EXTENDS",
      "source": "org.argouml.uml.diagram.ui.FigNodeModelElement",
      "target": "org.argouml.uml.diagram.use_case.ui.FigActor"
    },...
  ]
}
```

Listing 1: Excerpt of the JSON file of ArgoUML-SPL

2.2 Automatic identification and visualization of implemented variability

*The **symfinder** toolchain.* By definition, a variation point (*vp*) with its variants represent the unchanged and changed parts in software design, are realized by an implementation technique, and abstract the structure (*a.k.a.*, design) and the functionality of the implemented variability [30]. Their uniform identification is desirable as diverse object-oriented techniques can be used within an SPL [29]. Towards understanding the implemented variability of an SPL realized with such techniques within a single code base,

we provided in a previous work [24, 30] an approach for identifying and visualizing *vp*-s with variants of an SPL in a uniform way. The different kinds of *vp*-s with variants are interpreted in terms of local symmetry [32], as a common property among the traditional variability implementation techniques and can then be automatically detected. The associated toolchain, *symfinder*, was used to automate and validate the approach on eight realistic and open-source variability-rich systems, including ArgoUML-SPL. Specifically, *symfinder* parses its code base and identifies *vp*-s with variants by simply detecting symmetry in up to seven variability implementation techniques at class and method level [30], namely the symmetry in class as type, class subtyping, method overloading, factory pattern, strategy pattern, template pattern, and decorator pattern. The identified *vp*-s with variants data are stored with other extracted information into a Neo4j database, and a JSON file can be produced for reuse, such as in a web-based visualization [24].

*Application of **symfinder** to ArgoUML-SPL.* For this study we run *symfinder* on the same `src/argouml-app` package of ArgoUML-SPL [5] as for the ground truth, by using the commit ID given in Table 2. The identified *vp*-s with variants and their visualization for the ArgoUML-SPL are publicly available [6]. An excerpt from the resulting JSON file is shown in Listing 1. It contains an array of `nodes` and an array of `links`. Each `node` represents a class being a *vp* or a variant and possesses multiple attributes. For instance, `org.argouml.uml.diagram.ui.FigNodeModelElement` is the *vp*'s name which has 18 class variants, 1 *vp* with 3 variants at constructor level, and 2 other *vp*-s with 5 variants at method level. The `types` attribute characterizes properties of the class that the node represents, such as its class or interface nature, the presence of a *vp* or variant, or the presence of design patterns. Whereas, a `type` in the `links` represents the relation of this *vp* with its variants, such as with the `org.argouml.uml.diagram.ui.FigActor` variant. In general, each link represents a superclass inheritance or an interface implementation relationship, with its `source` being the superclass/interface and its `target` being the subclass/implementation.

Figure 2 shows an excerpt from the resulting visualization for the analysed ArgoUML-SPL. As an example, it contains the same illustrative *vp* given in Listing 1, `ui.FigNodeModelElement`, surrounded by its class level variants, such as the `ui.FigActor` variant. Different node types represent the used technique to implement the *vp*-s, as explained in the caption of Figure 2, and can be shown through the `'Show legend'` menu. Contrary to a feature model that has a tree-like structure (*cf.* Figure 1), it can be noted that the identified *vp*-s with variants have a forest-like structure. As shown in Figure 2, some class level *vp*-s are solitary, without class level variants, or are part of a larger tree, with class level variants.

Resulted vp-s with variants in ArgoUML-SPL. In this work, we use the JSON file of ArgoUML-SPL to calculate its identified number of class and method level *vp*-s with variants, given in Table 2. It shows that in ArgoUML-SPL 1,560 *vp*-s with variants are identified at class level, some of which have *vp*-s with variants at method level. However only those at class level have a `name` attribute, for instance, the *vp* `ui.FigNodeModelElement` in Listing 1. From the

[5] https://github.com/marcusvnac/argouml-spl
[6] https://deathstar3.github.io/symfinder-demo/vamos2020/

Table 2: The resulting #vp-s and #variants, at class and method level, for the analysed commit ID of ArgoUML-SPL by using the *symfinder* tool

Analysed package:	**src/argouml-app**		
Commit ID:	*bcae37308b13b7ee62da0867a77d21a0141a0f18*		
Class level:	#variation points (#vp-s)		200
	#vp-s / #variants		258
	#nodes with method level vp-s		107
	#variants		995
#*vp*-s/#variants:	**Solitary: 154**	**Tree: 1,406**	
Class level total:			**1,560**
Method level:	#variation points (#vp-s)		631
	#variants		1,551
Method level total:			**2,182**
Total:	**#vp-s: 1,089**	**#variants: 2,653**	

Table 3: The mapping of an identified *vp* with its eight variants at class level to features, visualized also in Figure 2

A *vp* with variants	**Feature**
vp: **`ui.FigNodeModelElement`**	Cognitive, Logging
`use_case.ui.FigActor`	Use Case
`sequence.ui.FigClassifierRole`	Sequence
`static_structure.ui.FigComment`	Logging
`collaboration.ui.FigClassifierRole`	Collaboration
`activity.ui.FigObjectFlowState`	Activity
`ui.FigEdgePort`	-
`deployment.ui.FigObject`	Deployment
`activity.ui.FigPartition`	Activity

1,560 *vp*-s and variants at class level, 154 are solitary *vp*-s whereas the rest 1,406 are part of a larger tree. All of them in Table 2 are *potential vp*-s with variants, meaning that they may implement domain features. To evaluate their potential of being variability places in code assets, we manually map them to features and observe the results in the next sections, including the potential to automate and to use visualization for this mapping in the future.

3 EVALUATING THE RELEVANCE OF *vp*-s WITH variants

In order to evaluate the relevance of the *vp*-s with variants identified by the *symfinder* approach, we undertake two steps. First, we manually map the identified *vp*-s with variants of ArgoUML-SPL to its domain features. Then, we evaluate their relevance by adapting two well-known measures, namely *precision* and *recall*.

3.1 Data normalization

Before the mapping process, we normalized the granularity of traces for the domain features with the granularity of the identified *vp*-s with variants, so they all become of a common class level granularity. Specifically, whenever a feature in the ground truth had

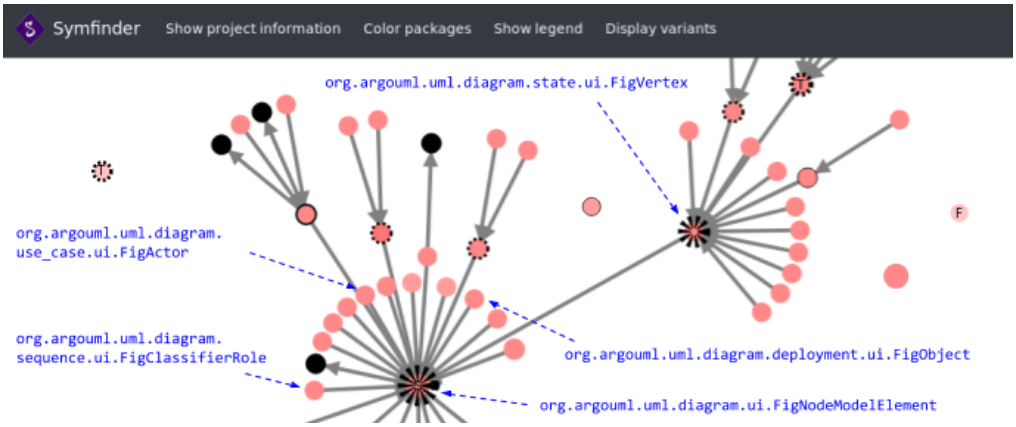

Figure 2: Excerpt of the visualization of identified *vp*-s in ArgoUML generated by *symfinder*. Annotations in blue show potential *vp*-s names that are displayed when hovering a node. Legend: ● - Class as type (*vp* or variant), ● - Class variant with inner *vp*-s, ● - Abstract class (*vp*), ● - Interface (*vp*), ● - Constructor overloading (*vp*), ● - Method overloading (*vp*), Node with 'F', 'S', 'T', 'D' symbol - Factory, Strategy, Template, or Decorator pattern, ⟶ Inheritance relationship

one of its traces to a class refinement, complete method, or method refinement, we simplified that trace to the whole class. For instance, the feature `State` in the ground truth had one of the trace links to `org.argouml.ui.cmd.GenericArgoMenuBar initMenuCreate() Refinement` [7], which is a trace at the statement level within the method `initMenuCreate()`. In such a case, we truncated the trace to the whole class `org.argouml.ui.cmd.GenericArgoMenuBar`. This means that we consider all features' traces, but we only change their granularity to class level. Then, from the identified *vp*-s with variants, we considered only those at the class level, specified in Table 2, but these include also all method level *vp*-s with variants. This normalization is necessary for two reasons. First, *symfinder* records the names only for class level *vp*-s and variants, whereas for those at method level it records only their total number (*cf.* Listing 1). The second reason is that *vp*-s with variants are related only to the structural elements in code, such as classes or methods for now, whereas features in the ground truth have traces mostly to their refinements, where about 73% of them are at statement level.

3.2 Mapping of variabilities

After aligning their granularity, we can process with the mapping of *vp*-s with variants to features by using their traces. We have 712 traces (without replication) for the 8 optional features, including 14 of their combinations, and 2 of their negations (*cf.* Table 1). Toward simplifying the mapping process we first exported features with their trace links and *vp*-s with variants into an Excel file. Then, for each feature, each of its trace links is looked up if it is a *vp* or variant, by using the `VLOOKUP` function in Excel. Whenever a *vp* or variant is associated with a feature's trace link, it is marked as a relevant *vp* or variant. The whole mapping process is conducted by one person, then it is double-checked with a second person. In this way, we use these traces to map features manually to the 458 *vp*-s, 107 classes with method level *vp*-s, and 995 variants at the class level (*cf.* Table 2). For instance, in Table 3 is given the mapping of the illustrative *vp* `ui.FigNodeModelElement` and eight from its eighteen class variants (*cf.* Listing 1 and Figure 2). The *vp* itself maps to two features, `Cognitive` and `Logging`, and the eight shown variants map to six features. Seven of the shown variants are mapped to one feature, two of which map to the same `Activity` feature, whereas one variant, `ui.FigEdgePort`, is without a mapping, meaning that this variant does not appear as a trace link in any of the features in the ArgoUML-SPL's ground truth.

To summarize, Figure 3 shows the number of *vp*-s and variants mapped to 8 optional features, whereas Figure 4 shows the range of *vp*-s and variants mapped to their 14 feature combinations and 2 feature negations. In total there are 163 *vp*-s and 613 variants, or 593 of them without duplication among features, which we refer to as *relevant* *vp*-s and variants.

By simply analysing these data, we notice that about 89% of the relevant *vp*-s and variants are mapped to the 8 features, whereas 11% of them are mapped to the 14 feature combinations and 2 feature negations. Then, from Figure 4, the majority of feature combinations and negations are mapped only to variants, about 90% of nodes involved in this mapping being variants.

3.3 Mapping measures

As expected from a non-trivial mapping, several of the identified *vp*-s and variants are without a mapping to features in the ground truth, and conversely. Specifically, from the 1,560 class level *vp*-s and variants that are identified in ArgoUML-SPL (*cf.* Table 2), 593 of them have a mapping to at least one feature. This means that there are 967 *vp*-s and/or variants that are without a mapping to features. Then, from all 712 features' traces in the ground truth, 119 of them are not used for the mapping of *vp*-s or variants. At the feature level, there is only the `not_Cognitive` feature negation that has a single trace and is without a mapping to *vp*-s or variants.

Therefore, in order to evaluate more accurately the relevance of the identified *vp*-s and variants for feature mapping, we define *precision* and *recall* measures in our specific context.

[7] https://github.com/but4reuse/argouml-spl-benchmark/blob/master/ArgoUMLSPLBenchmark/groundTruth/STATEDIAGRAM.txt

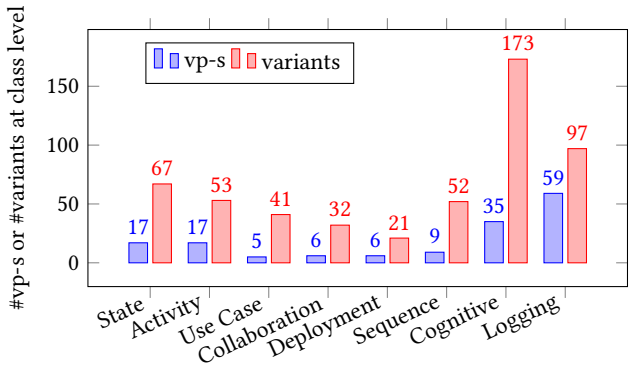

Figure 3: The #vp-s and #variants mapped to eight features

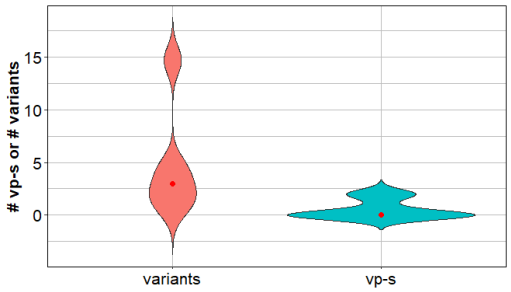

Figure 4: The overall range of *vp*-s and variants mapped to the fourteen feature combinations and two feature negations given in Table 1

Precision. Let be T_{gt} the set of traces for all features given in the ground truth and I_{vp-v} the set of all identified *vp*-s and variants by *symfinder*. We use *precision* to measure the percentage of the identified *vp*-s and variants that are relevant for the feature mapping. Thus, for the current mapping in ArgoUML-SPL, the *vp*-s and variants that are mapped to the features in the ground truth are true positives (TP), referred as relevant *vp*-s and variants, whereas the *vp*-s and variants without a mapping are false positives (FP), or irrelevant *vp*-s and variants. Therefore,

$$precision = \frac{TP}{TP + FP} = \frac{|T_{gt} \cap I_{vp-v}|}{|I_{vp-v}|} = \frac{593}{1560} = 0.3801$$

This means that about 38% of our identified *vp*-s and variants are relevant for the feature mapping, whereas the rest, 62% of them, are irrelevant. Such a low precision was expected for two main reasons. First, the eight optional features used to extract an ArgoUML-SPL are coarse grain features and are selected by authors based on the ArgoUML domain knowledge [8]. Thus, their study certainly misses some information regarding how complete is the list of ArgoUML-SPL features. This means that the ground truth may be incomplete, which explains the *vp*-s and variants without a mapping. Then, it is likely that not all of our identified places with a symmetry in code are variability related, such is the case with the usage of preprocessor directives in C/C++. Specifically, Zhang *et al.* [31] state that *"from our experience most #ifdef blocks (*e.g.*, 87.6% in the Danfoss SPL) are actually not variability related, but for other purposes such as include guards or macro substitution"*. This implies that in OO variability-rich systems, in addition to implementing variability, the technique of inheritance is mostly used for other reasons too (*e.g.*, in 62% of cases in the ArgoUML-SPL). Still, the high number of false positive *vp*-s and variants has especially an impact on the time spent to establish the mapping.

Recall. Through *recall* we measure the percentage of features' traces in the ground truth that are used for the mapping of *vp*-s and variants to features. Thus, the traces that are used for the mapping are true positives (TP), whereas those that are not used are false negatives (FN). Therefore,

$$recall = \frac{TP}{TP + FN} = \frac{|T_{gt} \cap I_{vp-v}|}{|T_{gt}|} = \frac{593}{712} = 0.8329$$

This means that about 83% of features' traces in the ground truth are used for the *vp*-s and variants mapping to features. Actually, we measured recall also for each feature. As a result, the 8 optional features have lower recall values compared with their feature combinations which have, except one, a 100% recall value.

By a deeper analysis, we noticed that the 17% of the unused features' traces usually refer to the statements within the initialization classes, such as `Main` classes, or use other external libraries. This clearly explains why they are not used for the mapping, as the initialization classes are not categorized as *vp*-s or variants by *symfinder*, and *vp*-s and variants coming from external libraries are also filtered out by it.

Summary. These two measures indicate that the identified *vp*-s with variants have a lower precision but are highly robust. This means that less than half of the identified *vp*-s with variants are relevant (38%), but they implement a high percentage (83%) of all given domain features. Those without a mapping are certainly because the available features are of a coarse grain, the ground truth may be incomplete with features, or some of the *vp*-s and variants are not variability related. Still, the high recall indicates the high relevance of the identified *vp*-s with variants in a variability-rich system, thus showing the feasibility of our approach by *symfinder*.

4 ADDITIONAL RESULTS

We further analysed the resulted relevant *vp*-s with variants and their mapping to features in ArgoUML-SPL. Our focus was to analyse *(i)* what are the main challenges for an automated mapping approach in this context, and *(ii)* whether the current visualization options by *symfinder* support well the mapping, being it automatic or not. These observations are given in the following.

4.1 Towards an automatic approach

Considering that the variability of ArgoUML-SPL is implemented using object-oriented traditional techniques where the main separation of concerns are not features, it is expected that domain features do not align well with the identified *vp*-s and variants, thus complexifying the mapping process. To confirm this, we analysed the

Table 4: The scattering degree of features: the number of *vp*-s and variants (#vp&v) used to implement a given feature

ID	Feature	#vp&v	ID	#vp&v	ID	#vp&v
[1]	State	84	[2 – 8]	3	[5 – 8]	4
[2]	Activity	70	[2 – 1]	7	[5 – 3]	1
[3]	Use Case	46	[7 – 5]	15	[8 – 6]	15
[4]	Collaboration	38	[7 – 8]	15	[8 – 1]	4
[5]	Deployment	27	[7 – 6]	3	[8 – 3]	4
[6]	Sequence	61	[4 – 8]	3	[6 – 1]	1
[7]	Cognitive	208	[4 – 8 – 6]	2	¬[7]	0
[8]	Logging	156	[4 – 6]	5	¬[8]	4

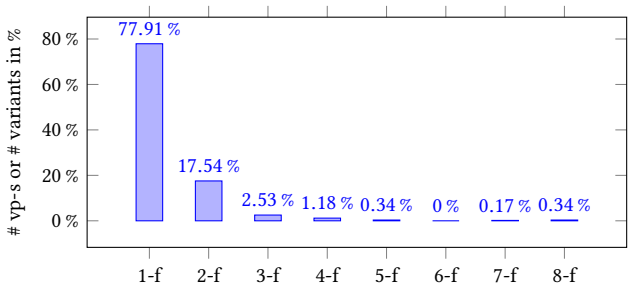

Figure 5: The #vp-s and #variants in % with a mapping from one feature (1-f) up to eight features (8-f), including to their combinations and negations

multiplicity of their mapping links by studying the crosscutting nature of optional features, including their combinations and negations. We measured the scattering and tangling degrees [10] of a given feature implemented by *vp*-s and/or variants. Then, we analysed whether *vp*-s and variants of a given feature share a part of their names, as it is extensively used as a first easy step towards an automatic mapping when conditional compilations [18] and feature modules [3] are used.

Crosscutting nature of features. The *scattering degree* is used to measure the number of *vp*-s and variants that are used to implement a given feature, — when a feature is scattered in code, finding automatically all places that implement it in an OO software system is more difficult —. In Table 4 is given the number of *vp*-s and variants used to implement the 8 optional features, their 14 combinations, and their 2 negations in ArgoUML-SPL. It shows that most of the features are implemented by more than one *vp* and/or variant, whereas two feature combinations are implemented by exactly one *vp* or variant, and one feature negation (¬[7] - `not_Cognitive`) is implemented by none of the identified *vp*-s or variants. This indicates that a feature can have zero, one, or more mapping links to *vp*-s and variants, thus their mapping is 1 to M.

The *tangling degree* is used to measure the number of features that are partially[8] implemented by a given *vp* or variant — whenever a *vp* or variant is used to implement more than one feature, finding all features that it addresses in an OO software system is

[8]We use *partially* as *vp*-s with variants are mostly a refinement of domain features [14].

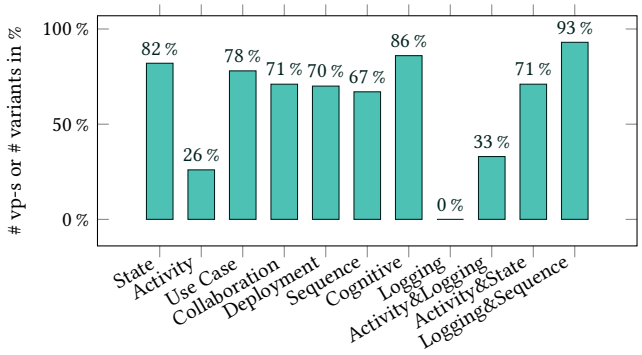

Figure 6: The relevant #vp-s and #variants in % that share a part of their name with features that they implement

also more difficult —. As there is a large number of *vp*-s and variants, we summarized the resulted tangling degree as in Figure 5. It shows that almost 78% of *vp*-s and/or variants are used to implement one of the features, or one of their combinations, or negations. Then, almost 18% of them implement two of the features, whereas those that implement between three and eight features in maximum are under 3%. Specifically, the highest tangling have exactly the 8 optional features, which have in common two *vp*-s and/or variants, or 0.34% of them. This indicates that different features are partially implemented by a given *vp* or variant [9]. Therefore, features have an N to 1 mapping in code assets.

Based on the scattering and tangling degree of features in Table 4 and Figure 5, the mapping between features and *vp*-s with variants is N to M. This observation confirms that multiple mapping links are required for a complete mapping, thus challenging the implementation of an automatic approach. Features highly crosscut also when, for instance, preprocessors in C/C++ are used. However, using similar names between features and preprocessor directives is known to simplify mapping automation [18]. This multiple mapping is also the main reason for feature modularization into modules with similar names with domain features [7].

Automatic mapping through name matching. Despite the multiple mapping links, we also analysed to what degree a given feature can be mapped automatically to their *vp*-s and variants by simply using their names. For name matching we used the pattern search algorithm [26] by using `Search` function in Excel. It is case-insensitive and searches a given pattern substring within a longer string and returns its location. For example, in the last variant `activity.ui.FigPartition` in Table 3 we searched for the `Activity` feature and the function returns a location, meaning that their names share the `'activity'` word. In Figure 6 we show the percentages of the relevant *vp*-s and variants that share a part of their name with the features that they implement in ArgoUML-SPL.

These results show that most of the *vp*-s with variants could be mapped automatically to features, for instance, 82% of *vp*-s and variants that implement `State` feature use the `'state'` text in their name. In a lower percentage are *vp*-s and variants for the `Activity` features, whereas for the `Logging` feature there is no *vp* or variant

[9]Here are considered only the relevant *vp*-s and variants.

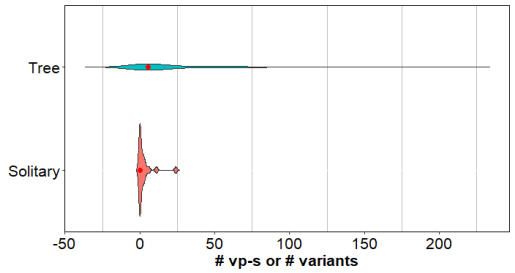

Figure 7: The range of solitary *vp*-s with variants and those within a tree used to implement the eight optional features, their fourteen combinations, and two negations

that uses the text `'logging'` in their name. In addition to the figures for the 8 optional features, we show in Figure 6 the percentages for only 3 feature combinations as the other combinations share their names with 100% of their *vp*-s and variants. Regarding feature negations, they are omitted from Figure 6 as they do not share their name with any of their *vp*-s or variants. In general, 74.12% of the relevant *vp*-s with variants share their names with the features that they implement. These results mean that, in this specific case, the mapping of *vp*-s with variants to features can be automated to some extent by simply relying on their naming convention.

4.2 A visualization that supports mapping

In some previous work [30] and Figure 2, we showed that the implemented variability of a variability-rich system has a forest-like structure. Usually, there are trees of different sizes of *vp*-s with variants, but there are also several solitary *vp*-s at class level. In [30], we relied on the center's theory [1] to assume that the density of *vp*-s with variants is a means for locating and describing the most intense places with variability in reusable code assets. This density can be discerned directly from the visualization that *symfinder* provides, such as the visualized tree of *vp*-s in ArgoUML-SPL with the `ui.FigNodeModelElement` *vp* and its variants in Figure 2. The visualization in *symfinder* has thus the option to filter out solitary *vp*-s, as places with the lowest density of variability. Moreover, to easily discern the zones of interest *w.r.t.* variability, *symfinder* visualizes all *vp*-s with variants at class level except those variants without method level variability.

In the ArgoUML-SPL case study, we analysed whether filtering out solitary *vp*-s and hiding class level variants without method variability from the visualization help to keep in focus those *vp*-s and variants that are relevant for the mapping. In which case, these two visualization options would support the feature mapping as the visualized information would be enough to help an expert in pinpointing the right *vp*-s and variants during a semiautomatic mapping process.

On filtering out solitary vp-s. In Table 2 are given the total identified number of solitary *vp*-s and the number of *vp*-s and variants that are part of a tree in ArgoUML-SPL. However, as stated in Section 3.3, only 593 of *vp*-s and variants, or 38% of them, are relevant for the mapping to features. We expect that these relevant *vp*-s and variants are not filtered out by *symfinder*.

To reason about whether filtering out solitary *vp*-s is hampering the mapping process, we anaysed how many of the relevant *vp*-s and variants are solitary and part of a larger tree. In Figure 7 are shown their resulted ranges [10]. It shows that most of the relevant *vp*-s and variants (92.53%) are part of a larger tree, whereas only a small number of them (7.47%) are solitary *vp*-s. Based on these results, we observed that places with a higher density in the visualization contain most of the relevant *vp*-s and variants. Therefore, filtering out solitary *vp*-s can be done without prejudicing a mapping process, as a small number of them have a contribution in implementing features.

However, during the analysis of the tangling degree of features, we noticed that one of the two *vp*-s that are used to partially implement all 8 optional features (*cf.* Figure 5) is a solitary *vp*. This means that, despite the small number of relevant solitary *vp*-s, they can be of a higher importance. Therefore, removing the filter for solitary *vp*-s is also needed — this option is available in *symfinder*—.

Displaying all variants. The current version of *symfinder* visualizes all identified *vp*-s, and only their variants that have a method level variability. Therefore, for the ArgoUML-SPL's feature mapping we extracted into a second JSON file all identified *vp*-s with variants, including those variants without method level variability.

To reason about whether *symfinder* should visualize these omitted variants in the future, we analysed how many of the class variants without method level variability are relevant for the mapping in ArgoUML-SPL. It resulted that there are 107 such variants, and only 32 of them are relevant for the mapping. Compared with the overall number, they represent 5.40% of the relevant *vp*-s and variants. Although the number of the relevant class variants without method level variability seems to be small, we consider that *symfinder* should contain in the visualization 100% of the relevant *vp*-s and variants. Surprisingly, we noticed that the second *vp* or variant that is used to partially implement all 8 optional features (*cf.* Figure 5) is a variant without method level variability. Therefore, in the last version of *symfinder* we improved its capabilities by adding a `'Display variants'` option (*cf.* Figure 2), which visualizes *vp*-s with all their variants.

4.3 Threats to validity

As we consider from the start a single case study, it is obvious that we cannot draw conclusions on the generalization of the results. The threats to validity we mainly face are then related to the normalized data and the observed results.

A first threat concerns the normalized granularity of features' traces with the granularity of *vp*-s and variants. Specifically, all features' traces in the ground truth are considered, but whenever a trace was at statement level we considered only its class. Then, although *symfinder* identifies and visualizes 3,742 *vp*-s with variants (*cf.* Table 2), we consider for mapping only the 1,560 of them that are at class level. But, almost all excluded method level *vp*-s with variants are within the considered class level *vp*-s with variants. Additionally, in case that a class has only method level variability, we also considered it (the 107 *#nodes with method level vp-s* in Table 2 are such classes). This normalization of data might have an

[10] The negative values are because violin plots rely on a *kernel density estimation*.

impact on how we interpret the feature mapping results and the relevant *vp*-s with variants, but we do not have indications that the number of the relevant *vp*-s and variants may change. Towards eliminating this data normalization, the method *vp*-s and variants names should be recorded by *symfinder* into the JSON file, which will be done in the near future.

A second threat to validity regards the presentation of different aspects of the results. For instance, we give the percentage of relevant *vp*-s and variants that could be automatically mapped to features by using their names. We kept out of analysis the 49 features' trace links (41.18%) that are not used for *vp*-s and variants mapping but they use feature names too. Then, in the current visualization, we display only names of *vp*-s and variants at class level. In case that we consider also those at method level, we need to come up with a way for displaying their names too, and to analyse how filtering out solitary *vp*-s with variants at class level impacts those that can be relevant at method level. Nevertheless, we believe the presented results are relevant enough to show that the *symfinder* variability identification and visualization approach is expressive enough to be used for a proper feature mapping process on the case study. Besides the complete raw and analysed data are available online [11], so they can be used and extended by the community.

5 RELATED WORK

To manage variability in SPLs, most of the existing approaches propose to modularize features into physically separate modules [3] or use conditional compilations, such as preprocessors in C/C++ [13, 18, 19, 28], or a form of annotations [8, 12]. In these cases, features have a straightforward mapping in code assets using their naming conventions. However, extensive manual effort is required to add annotations in code assets or refactor them into feature modules. With *symfinder* we provide an automatic approach for identifying variability places in OO code assets. The results of the manual feature mapping conducted in ArgoUML-SPL case study show that these automatically identified places are highly relevant and indeed implement domain features. Similarly to other approaches, this mapping is likely to be automated, although not completely, by simply using the features and *vp*-s naming.

Since Couto *et al.* [8] extracted the ArgoUML-SPL, it has been proposed [22] and extensively used [9, 21, 23] as a benchmark for reverse engineering variability and evaluating feature location techniques [4, 25]. In contrast, our variability identification and visualization approach is more a tool support for understanding implemented variability in forward engineering [30]. Thus, in addition to its usage in reverse engineering, we show that ArgoUML-SPL can also be used as an interesting study in another context.

Besides a recent mapping study shows that there are several approaches for information visualization in SPL engineering [20]. Only a few of them visualize the variability at code level. From them, the approach for a virtual separation of concerns [11, 16, 17] is mostly related to our visualization approach. Similarly, it is used for variability management and relies on a different color per feature to manually map them to code assets. In contrast, we use a graphical visualization of variability by using more visualization parameters, namely position, size, shape, value (lightness), color hue, orientation, and texture. We also automatically visualize the variability and keep it separate from code assets.

6 CONCLUSION

Many realistic variability-rich software systems are implemented using object-oriented techniques and do not follow a complete software product line approach. Towards managing their variability, we have previously proposed a tooled approach, *symfinder*, that automatically identifies and visualizes the variation points (*vp*-s) with variants in their code assets. While *symfinder* has been successfully applied to several systems, the mapping of *vp*-s with variants to domain features was not explored yet. In this paper we measured the relevance of the automatically identified *vp*-s and variants by observing their manual mapping to the domain features in ArgoUML-SPL. For experimentation we used the preexisting ArgoUML-SPL's ground truth on domain features and adapt the *precision* and *recall* measures to evaluate the relevance of the identified *vp*-s and variants. It resulted that less than half of the identified *vp*-s and variants are relevant (38%), but they implement a high percentage (83%) of all domain features. As the used ground truth is partial and our automatic identification does not cover all implementation techniques, this shows that the approach must be improved but is powerful enough to enable a successful mapping of features.

As a starting point for future works, we already provided the first observations towards an automatic mapping approach, using naming conventions, and did some first improvements in the *symfinder* visualization to support a mapping process. Moreover, *symfinder* is available [12] and can be used on other software systems, with a ground truth, to further improve the precision of our identification approach. For this, we plan to extend *symfinder* so that it also records the *vp*-s with variants at method level, which could then be considered within a mapping process. We also plan to improve the visualization in order to provide users with a more intuitive way to visualize the organization of variability implementations.

REFERENCES

[1] Christopher Alexander. 2002. *The nature of order: an essay on the art of building and the nature of the universe. Book 1, The phenomenon of life.* Center for Environmental Structure.

[2] Nicolas Anquetil, Uirá Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummler, and André Sousa. 2010. A model-driven traceability framework for software product lines. *Software & Systems Modeling* 9, 4 (2010), 427–451.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer.

[4] Wesley KG Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.

[5] Kathrin Berg, Judith Bishop, and Dirk Muthig. 2005. Tracing software product line variability: from problem to solution space. In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries.* South African Institute for Computer Scientists and Information Technologists, 182–191.

[6] Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. 2013. Systems and software variability management. *Concepts Tools and Experiences* (2013).

[7] José M Conejero and Juan Hernández. 2008. Analysis of crosscutting features in software product lines. In *Proceedings of the 13th international workshop on Early Aspects.* ACM, 3–10.

[8] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. 2011. Extracting software product lines: A case study using conditional compilation. In

[11] https://deathstar3.github.io/symfinder-demo/vamos2020.html

[12] https://github.com/DeathStar3/symfinder/tree/vamos2020

2011 15th European Conference on Software Maintenance and Reengineering. IEEE, 191–200.

[9] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques using ArgoUML-SPL. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 16.

[10] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology* 106 (2019), 1–30.

[11] Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. 2011. Using background colors to support program comprehension in software product lines. In *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*. IET, 66–75.

[12] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. 2012. A code tagging approach to software product line development. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 553–566.

[13] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering* 21, 2 (2016), 449–482.

[14] John M Hunt and John D McGregor. 2006. 9 Implementing a Variation Point: A Pattern Language. *Variability Management–Working with Variability Mechanisms* (2006), 83.

[15] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering* 5, 1 (1998), 143.

[16] Christian Kästner. 2010. Virtual separation of concerns-toward preprocessors 2.0/von Christian Kästner. (2010).

[17] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code. In *SPLC (2)*. 303–312.

[18] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating consistency between a feature model and its implementation. In *International Conference on Software Reuse*. Springer, 1–16.

[19] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 105–114.

[20] Roberto Erick Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. 2018. A systematic mapping study of information visualization for software product line engineering. *Journal of Software: Evolution and Process* 30, 2 (2018), e1912.

[21] Jabier Martinez, Wesley KG Assunção, and Tewfik Ziadi. 2017. ESPLA: A catalog of Extractive SPL Adoption case studies. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*. ACM, 38–41.

[22] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature location benchmark with argoUML SPL. In *Proceeedings of the 22nd International Conference on Systems and Software Product Line-Volume 1*. ACM, 257–263.

[23] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley KG Assunção, and Alexander Egyed. 2019. Comparison-based feature location in ArgoUML variants:[challenge solution]. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. ACM, 17.

[24] Johann Mortara, Xhevahire Tërnava, and Philippe Collet. 2019. symfinder: a toolchain for the identification and visualization of object-oriented variability implementations. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*. ACM, 56.

[25] Julia Rubin and Marsha Chechik. 2013. A survey of feature location techniques. In *Domain Engineering*. Springer, 29–58.

[26] Graham A Stephen. 1994. *String searching algorithms*. World Scientific.

[27] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. 2005. A taxonomy of variability realization techniques. *Software: Practice and Experience* 35, 8 (2005), 705–754.

[28] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 531–551.

[29] Xhevahire Tërnava and Philippe Collet. 2017. On the diversity of capturing variability at the implementation level. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*. ACM, 81–88.

[30] Xhevahire Tërnava, Johann Mortara, and Philippe Collet. 2019. Identifying and visualizing variability in object-oriented variability-rich systems. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. ACM, 32.

[31] Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. 2013. Variability evolution and erosion in industrial product lines: a case study. In *Proceedings of the 17th International Software Product Line Conference*. ACM, 168–177.

[32] Liping Zhao and James Coplien. 2003. Understanding symmetry in object-oriented languages. *Journal of Object Technology* 2, 5 (2003), 123–134.

[33] Liping Zhao and James O Coplien. 2002. Symmetry in class and type hierarchy. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Australian Computer Society, Inc., 181–189.