Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships

Johann Mortara, Xhevahire Tërnava, Philippe Collet, Anne-Marie Dery-Pinna

Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships

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

Xhevahire Tërnava
xhevahire.ternava@irisa.fr
Université de Rennes 1, INRIA/IRISA
France

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

Anne-Marie Pinna-Dery
anne-marie.pinna@univ-cotedazur.fr
Université Côte d'Azur, CNRS, I3S
France

ABSTRACT

Many variability-rich object-oriented systems rely on multiple traditional techniques (inheritance, patterns) to implement their variability in a single codebase. These variability implementation places are neither explicit nor documented, hampering their detection and variability comprehension. Based on the identification of symmetry property in seven implementation techniques, a first approach was proposed with *symfinder* to automatically identify and display the variability of a system in a graph-based visualization structured by inheritance. However, composition, or more generally the usage relationship, is extensively used to implement the variability in object-oriented systems, and without this information, comprehending the large amount of variability identified by *symfinder* is not trivial. In this paper, we present *symfinder-2*, an extension of the former approach that incorporates the usage relationships to better identify potential variability implementations. We provide two ways to mark classes as entry points, user-defined and automatic, so that the visualization is filtered and enables users to have a better focus when they identify variability. We also report on the evaluation of this extension to ten open-source Java-based systems.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software reverse engineering**; *Object oriented architectures.*

KEYWORDS

variability identification, variability visualization, variability-rich object-oriented software systems, *symfinder*

1 INTRODUCTION

Whatever is their size, most modern object-oriented software systems are variability-rich [11]. The variability among the software variants of these systems is often implemented in a single codebase using traditional techniques, such as inheritance, parameters, overloading, or software design patterns [10, 38]. As they do not follow a fully-fledged software product line approach [4], their domain variability (*i.e.*, features) is hardly documented or made explicit to code assets. This directly hampers the comprehension and maintainability of the implemented variability [29].

Several manual and automatic approaches can be found in the literature for locating, identifying, and visualizing domain features in code [20, 25, 32]. But, in these approaches, features are either required to be known in advance or to be identified through reengineering a set of clone-and-own or legacy software systems into a product line. Actually, considering the single codebase systems, there was a lack of approaches to identify variation points (*vp*-s) with variants [1] implemented by object-oriented techniques [21], but also a lack of solutions to visualize them [20]. Complimentary to existing reverse engineering approaches, a new variability identification support was proposed with *symfinder* [37], which handles variability-rich object-oriented systems developed in a single codebase. More recently, Michelon et al. [24] also proposed a hybrid technique using runtime trace generation on *variants* from the same codebase to locate features, showing the interest of the research community in the context of single codebase systems.

On its side, *symfinder* provides automatic identification and visualization of potential *vp*-s with *variants* in object-oriented code assets, with support for the Java and C++ languages [26, 27]. Its identification approach of *potential vp*-s with variants is using the property of *symmetry* in their implementation techniques, for example, inheritance defines a substitution symmetry between the immutable part of the superclass (the *vp*) and the possible changes in its subclasses (the variants). *symfinder* also provides a visual graph-based representation of the identified variabilities, in which *vp*-s and variants are nodes related by their inheritance links while visual parameters, such as size and colour intensity, vary according to the number of symmetries in the method level. The aim is to make zones with many symmetries easily distinguishable. *symfinder* has been successfully applied to several real Java and C++ variability-rich systems [26, 28, 37], showing that a large part of the identified potential *vp*-s and variants actually implemented domain features [26], that is, they are actual *vp*-s with variants.

However, as more experiments and applications were conducted with *symfinder*, some drawbacks also appeared, especially on large systems. First, the visualization provided by *symfinder* exhibits *vp*-s and variants with heavy usage of inheritance, while some of the variability relies also on other mechanisms and are thus less visible. When some *vp*-s or variants are using others (*e.g.*, through attributes or parameters), this usage relationship is not exploited while it should create denser zones of variability and help in better structuring the variability visualization of large systems. Second, the user feedback shows that one tends to start an analysis

[1] Their definition is given in Section 2.1.

through some well-known entry point classes, being either exposed facade classes or a well-defined *application programming interface* (API) [30] that publish the customizable functionalities. However, *symfinder* did not provide any means to browse the identified variability from these entry points.

In the following, we first provide some background on the concept of symmetry used and the *symfinder* principles (Section 2). We then analyse the issues to be tackled (Section 3) and describe the different extensions made on the identification and visualization parts to build a new version of *symfinder* named *symfinder-2* (Section 4). During the identification of potential variation points with variants, we take into account their *usage relationships* so to display them in the visualization. We also refine the visualization so that entry point classes of a targeted system, such as API classes, can be used to find more easily the important zones of variability. A *parameterized density* metric can also help in filtering the visualization when no entry points are given. We applied *symfinder-2* to ten Java-based variability-rich systems and observed the impact of our changes and done improvements (Section 5). We notably evaluate the visualized graph, the remaining *vp*-s and variants, and the scalability of our extension. We finally discuss related work (Section 6), and conclude while evoking future work (Section 7).

2 BACKGROUND

In this section, we introduce the concept of symmetry in implementation techniques, *vp*-s with variants, their density, and *symfinder*.

2.1 Symmetry in object-oriented techniques

From the natural sciences, the symmetry of an object is defined as *the immunity to a possible change* [31]. Several studies have shown that most of the object-oriented techniques, such as inheritance, overloading, including software design patterns, can also be interpreted in terms of symmetry [5, 39]. Considering a whole codebase, these techniques can be seen as local symmetries [37], which allow a part of code to change while another part remains unchanged.

To illustrate these symmetries, let us consider an illustrative example from JFreeChart[2], a variability-rich object-oriented library that provides a family of charts, such as pie and meter charts, given in Figure 1. In this library, the abstract class `Plot` is the common part between the related variations of `PiePlot` and `MeterPlot`. These are some of the variations that make it possible to differentiate the *variants* of JFreeChart objects at the implementation level. The *common* and *variation* parts are commonly abstracted in terms of *variation points* (*vp*-s) and *variants*, respectively [6, 15]. In our example, the class `Plot` is a *vp* with two variants, `PiePlot` and `MeterPlot`. They can be abstracted as `vp_Plot`, `v_PiePlot`, and `v_MeterPlot`, respectively. Following the symmetry definition, inheritance defines a *substitution symmetry* for its subtypes. Here, the *possibility of a change* in the superclass `Plot` materializes in its potential different subtypes, such as `PiePlot` and `MeterPlot`, which vary regarding the way how they draw a chart. Still, they also *preserve* and conform to the common behaviour of their superclass.

Considering this way of reasoning, which is also illustrated in Figure 1, most of the object-oriented variability implementation techniques, including design patterns, can be described in terms of symmetry [5, 39]. At an abstract level, a *vp* represents the *unchanged* part while its variants are the *changed* parts in code assets of a system [37].

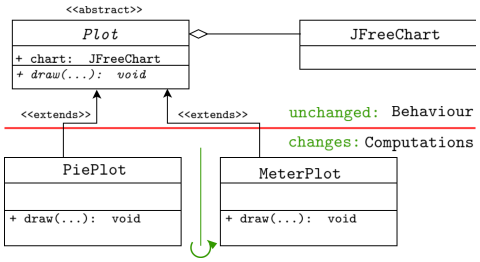

Figure 1: Inheritance as a local symmetry in JFreeChart

2.2 The *symfinder* approach

In object-oriented systems, the domain variability is hardly documented and traced to code assets (*e.g.*, in terms of features in a feature model [16]), as they are mainly not organized as fully-fledged product lines [2]. Consequently, their existing *vp*-s with variants in code assets, as in Figure 1, are neither explicit nor documented. In the literature, there are several manual and automatic approaches for locating, identifying, and visualizing domain features in code [7, 20, 24, 25]. In these approaches, features are either required to be known in advance or to be identified through reengineering a set of clone-and-own or legacy software systems into a product line. Unlike them, our recent tooled approach named *symfinder* [26, 27, 37] focuses on identifying the variability implementations in single codebase object-oriented systems. It provides automatic identification and visualization of potential *vp*-s with *variants* in code assets of a Java or C++-based variability-rich system [3]. The identification approach of *potential vp*-s with variants is based on the identification of local symmetries in seven object-oriented variability implementation techniques, namely *class as type*, *class subtyping*, *method and constructor overloading*, *strategy*, *template*, *decorator*, and *factory patterns*.

symfinder provides a visual representation of the potential *vp*-s and variants. Figure 2 shows the visualization of identified potential *vp*-s with variants for the illustrative example of JFreeChart given in Figure 1. Each class level *vp* or variant is visualized as a circle node that points out the used implementation technique, edges show their inheritance relationship, then the size and shades of the red colour of nodes indicate the number of method level *vp*-s with variants [4]. We follow here the visual principles of preattentive perception [12], using some of the seven parameters that can vary in visualization in order to represent data. The visualized variability then forms a disconnected graph based on inheritance links, while the visual representation of the metrics associated with each node (*e.g.*, size, colour intensity) creates zones that can be easily distinguished by their different density of symmetries [5]. For

[2] http://www.jfree.org/jfreechart/

[3] In this paper, we extend only the support for Java-based systems.

[4] `PiePlot` is a variant of `Plot`, however it is also a *vp* as it has two variants, thus is described as `vp_PiePlot` on the figure.

[5] It should be noted that the identification is based on symmetry in implementation techniques while the visualization on their density. Hence, one should not confuse the two and expect any kind of symmetry in the visualization.

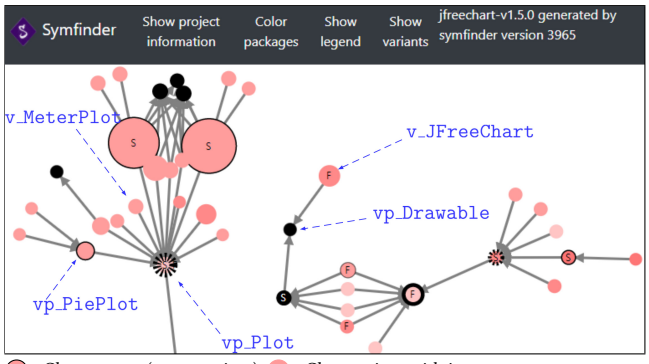

Figure 2: Identified *vp*-s and variants using *symfinder* for the excerpt of JFreeChart given in Figure 1.

example, the left part of Figure 2 is denser than the right part [6]. *The reader can note that we use vp-s and variants at the conceptual level while using sometimes classes when we discuss the technical identification parts, as well as nodes as the representation of classes in the visualization.* In addition, *symfinder* provides several other options for facilitating the comprehension of the variability of a system: filtering out the solitary nodes, showing or hiding class variants, metrics on identified *vp*-s and variants at different levels, coloring *vp*-s with variants that belong to the same package, and information on *vp* or variant code assets when hovering [7].

3 PROBLEM STATEMENT

symfinder has been applied to several Java and C++ variability-rich systems [26, 37], showing that it can successfully identify the potential variability in their code assets. It also helped to comprehend this variability by using the generated visualization. Moreover, an application on the ArgoUML-SPL [22] has demonstrated that a large part of the identified *vp*-s and variants by *symfinder* actually implement the ArgoUML's reverse-engineered domain features [26]. However, as more experiments and applications were conducted with *symfinder*, two issues also appeared.

Issue 1. Identifying inheritance relationships is not enough. By construction, the identified *vp*-s with variants with heavy usage of inheritance are the most visible in the *symfinder* visualization, that is, those implemented by classes that are represented as nodes. They are simply grouped with this relationship and the other visualized indicators, such as the size and colour intensity of the nodes. While the experiments show that real *vp*-s and variants were successfully identified with *symfinder*, it also appeared that their visualized density based only on the inheritance relationship is not enough to comprehend the variability of a system. In all studied systems, we identified many cases in which subclasses being variants were using other *vp*-s and variants. For example, in JFreeChart, `CompassPlot`, a variant of the *vp* `Plot`, uses every variant of the *vp* `MeterNeedle`, but they are displayed as disconnected in the visualization.

Besides, as studied by Zhao and Coplien [39], reusability in object-oriented systems is about the *instantiation* of templates, *composition* of the instances, and *substitution* of the instances. *symfinder* clearly identifies substitution of the instances, as for instance inheritance has a substitution symmetry [37], but the composition of other instances is not taken into account while it is used for reuse of code assets and implementation of design patterns. In the following, we will consider this relationship in the broad sense of the term, as a *usage relationship*, encompassing cases where a class uses another class in attributes and parameters of its methods.

Issue 2. Entry points are missing. With its zooming, filtering, and hovering capabilities, *symfinder*'s visualization naturally relies on the Shneiderman visual information seeking mantra [36]: *overview first, zoom and filter, then details on demand.* Nevertheless, in many systems and especially in large ones, users gave us feedback that they were missing some clear entry points to start browsing the visualization. The vast majority of studied systems indeed exposed facade classes, which were natural entry points expected by the users, or even a well-defined *application programming interface* (API) [30] where their reusable and customizable functionalities are textually exposed. The second issue we thus identify in this paper is the need for entry points to be specified and exploited in the visualization to facilitate variability comprehension.

4 *symfinder-2*

To address the identified issues, we extended the variability identification and visualization of *symfinder*, creating *symfinder-2*. Its sources and conducted experiments are publicly available [8] [9].

4.1 Handling the usage relationships

To address *Issue 1*, we extended the identification step by also identifying the *usage relationship*s among the identified *vp*-s with variants in a codebase. At parsing time, each usage relationship is identified and added to the *symfinder-2* graph database. Specifically, we consider that `class A` *uses* another `class B` if `class B` is used as a field type or method parameter type in `class A` [9]. For instance, in the example of JFreeChart in Figure 1, class `Plot` is used by class `JFreeChart` as it is referenced by a field inside the `Plot` class. Therefore, each of these classes will be identified by *symfinder-2* as either *vp* or variant and will be visualized with their inheritance and usage relationships.

Visually, we show in *symfinder-2* both inheritance and usage relationships, where inheritance relationships are grey arrows and usage relationships are represented as dashed (light) blue arrows. For example, the new visualization for the JFreeChart example, given in Figure 1, is shown in Figure 3. In comparison with its prior visualization in *symfinder*, given in Figure 2, now both relationships are represented. In particular, the usage relationship between `Plot` and `JFreeChart` is now explicit in Figure 3.

[6] The whole visualization for the JFreeChart 1.5.0 system is available online: https://deathstar3.github.io/symfinder2-demo/visualizations/jfreechart-v1.5.0.html
[7] Blue names and arrows in Figure 2 have been manually added for illustration.

[8] *symfinder-2*'s website: https://deathstar3.github.io/symfinder2-demo/
[9] https://doi.org/10.5281/zenodo.4946729

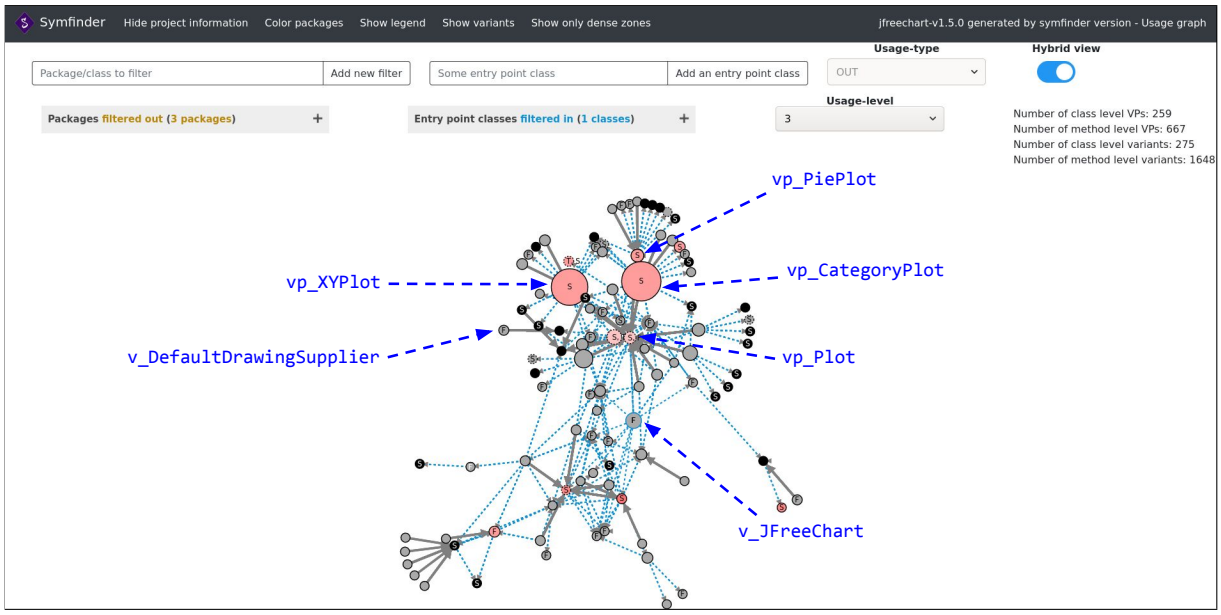

Figure 3: The identified *vp*-s and variants using *symfinder-2* for the excerpt of JFreeChart given in Figure 1

4.2 Handling the entry points

To address *Issue 2*, we extended the visualization to provide some entry points from which users can begin the exploration of the identified variability in a targeted system. In the following, we present two kinds of entry points that are available in *symfinder-2*.

User-defined entry points. We first added the possibility to filter and visualize only those potential *vp*-s with variants of a system that are under analysis by a user. As shown in Figure 3, one can choose a specific classpath and use it to refine the identified variability in its visualization. *symfinder-2* will show only those potential *vp*-s with variants and their interconnected *vp*-s or variants that are under the inheritance and usage relationships. For example, from the shown data in Figure 3, 926 potential *vp*-s at class and method levels are identified in JFreeChart.

Secondly, we added the possibility to filter in the potential *vp*-s with variants based on the *direction* or *level* of their usage relationships (*cf.* `'usage-type'` and `'usage-level'` boxes as in Figure 3). For example, the usage type `OUT` filters in only those class level *vp*-s and variants that use other *vp*-s or variants. Then, a usage level of 4 will display all *vp*-s and variants that have a usage relationship to one of the *vp*-s or variants through at most 4 identified usages transitively.

Lastly, we added the possibility to filter out the less dense zones with variability in the visualization. To achieve this filtering, a user needs to set a density threshold of potential *vp*-s with variants, which density is based on their usage and inheritance relationships [10]. For instance, the visualization from JFreeChart given in Figure 2 shows two places with different densities of potential *vp*-s and variants. The left one seems denser than the right one with the *inheritance-only symfinder*, whereas when usage relationships are considered using *symfinder-2*, both of these places are interrelated and create a new denser zone of potential *vp*-s and variants given in Figure 3. The density threshold is configured through setting two parameters in *symfinder-2*'s configuration file: (1) the minimum number of variants that a potential *vp* has, and (2) the maximum number of usage relationships between potential *vp*-s. These two values are used in queries over the graph database to identify the local symmetries that are under or above the given threshold. Finally, *symfinder-2* generates a new visualization where the potential *vp*-s and variants that are under the threshold are coloured in light grey and can be further excluded (`'Show only dense zones'` in Figure 3). As an example, in JFreeChart we set a threshold with the number of variants in a potential *vp* ≥ 5 and the usage relationships between the potential *vp*-s or variants ≤ 3. Hence, the `v_DefaultDrawingSupplier` variant is highlighted in grey as its `vp_DrawingSupplier` *vp* has less than 5 variants.

Automatically defined entry points. Another kind of entry points that we implemented in *symfinder-2* is the automatic filtering of potential *vp*-s and variants that are advertised as being part of an API. Actually, *symfinder-2* identifies classes annotated by the API Guardian library [11] as specific API entry points. The visualization then automatically adds all identified *vp*-s and variants as entry points so that they can be used to refine it as described in the previous paragraph. These added capabilities are made available to help users to chose interesting entry points in the visualization and to comprehend progressively the whole identified variability of a given system.

5 EVALUATION

In this section, we first introduce the subject systems and then define the research questions to evaluate the extended approach

[10] Due to page limitations, a detailed definition and implementation of the density threshold is given here: https://deathstar3.github.io/symfinder2-demo/density.html

[11] https://github.com/apiguardian-team/apiguardian

Table 1: The ten variability-rich subject systems. API: D - documented or A - annotated. Type: L - library, F - framework, and A - application

Subject system	Commit	LoC	# *vp*-s	# *variants*	API / Type
Java AWT	3319fcb	69,974	795	1,706	D / L
Apache CXF	4da7b71	48,655	3,403	7,625	D / F
JUnit	60aaf96	7,717	109	245	D / F
Maven	97c98ec	105,342	612	1,147	D / A
JFreeChart	1f6a91f	94,384	926	1,923	D / L
ArgoUML	d135342	134,367	776	1,959	D / A
Cucumber	323f724	42,662	238	282	A / F
Logbook	f0f36e7	16,210	96	162	A / L
Riptide	48b03a7	12,626	102	218	A / L
NetBeans	cade258	5,058,448	3,621	6,736	D / A

of *symfinder-2*. The *symfinder-2* and the used data to obtain the presented results are available online in a reproduction package [12].

5.1 Subject systems

To evaluate *symfinder-2*, we chose ten popular variability-rich subject systems, being Java applications, frameworks, or libraries (*cf.* Table 1). For the time frame of up to twelve last years, they have received between 150 and 8,000 stars in GitHub, but we particularly considered them because of the following criteria. The first six ones were already used to evaluate the first version of *symfinder* [37], namely Java AWT, Apache CXF, JUnit, Apache Maven, JFreeChart, and ArgoUML. Then, we chose the other three as they use in their codebase a form of API annotations, namely the API Guardian library, to annotate each code unit that constitutes their API. These new chosen systems are Cucumber – a framework for BDD testing, Logbook – a library to enable logging for different client- and server-side technologies, and Riptide – a library based on Spring to implement client-side response routing. Finally, we selected the NetBeans IDE because of its size with about 5 *M* lines of code (LoC), which helps in evaluating the scalability issues of both the approach and the prototyped toolchain.

5.2 RQ_1: Improved visualization

The visualization generated by *symfinder* shows the class level *vp*-s with variants as nodes, which are linked together through inheritance relationships. Whereas with *symfinder-2*, they are linked together through inheritance and usage relationships. In both cases, we observed that they form disconnected graph structures, which looks slightly different. Therefore, we defined RQ_1 :**does the identification of usage relationships have changed the variability visualization of a given system by *symfinder-2*?** To this end, we applied *symfinder* and *symfinder-2* to all ten subject systems and compared their respective visualizations. Results are given in Table 2.

On all studied systems, we notice a smaller number of disconnected graphs and isolated nodes by *symfinder-2* for the same number of nodes displayed, meaning that zones in the visualization

[12]https://zenodo.org/record/4946730

Table 2: Comparison of the number of disconnected graphs and isolated nodes with *symfinder* and *symfinder-2*

Subject	Nodes	*symfinder*		*symfinder-2*	
		Graphs	Isolated	Graphs	Isolated
Java AWT	431	55	142	2	20
Apache CXF	3085	473	1149	105	500
JUnit	118	23	36	6	18
Maven	616	177	172	21	79
JFreeChart	578	54	167	5	51
ArgoUML	1270	123	460	38	183
Cucumber	331	45	122	14	50
Logbook	117	19	40	4	16
Riptide	89	20	37	8	19
NetBeans	3498	504	1666	195	836

that seemed previously uncorrelated are now linked through usage relationships and appear as such on the new visualization. We observe that the difference between the number of disconnected graphs is not proportional to the size of the studied system. For instance, the number of NetBeans' graphs are reduced by 61% whereas JFreeChart's graphs are reduced by 90%. However, their number could be related to the architecture of the project. A project of an important size may have an architecture in layers, limiting the number of interactions between classes, and therefore exhibit fewer usage relationships. Besides, we notice that some isolated nodes still appear on *symfinder-2*'s visualization. This may suggest that other usage mechanisms are present in the studied systems [17]. Taking into account these specific types of usage relationships is part of our future work. Further, it is important to emphasize that although the visualizations by *symfinder* and *symfinder-2* have different numbers of disconnected graphs, their overall number and kinds of identified *vp*-s with variants remain unchanged. This indicates that *symfinder-2* is an extension of *symfinder* with an intact variability identification. To conclude, the reduced number of disconnected graphs by *symfinder-2* shows an improved and denser visualization of the identified *vp*-s and variants for a given system.

5.3 RQ_2: Starting density threshold

In addition to the two basic user-defined variability filtering capabilities that are added in the visualization and can be activated interactively, the third filtering capability is based on the density of potential *vp*-s and variants. As explained in Section 4.2, our filtering by density has for objective to reduce the number of potential *vp*-s and variants (*a.k.a.*, nodes) visible on the visualization through identifying those *vp*-s that have a minimum number of variants or those that are linked to another *vp* through a maximum number of usage relationships. This threshold is set before their identification, and setting it is not trivial as the user needs to know which is the right threshold to start with. Therefore, we defined RQ_2 : **what is the starting density threshold to begin with the comprehension of the visualized variability by *symfinder-2*?** To this end, we run *symfinder-2* on each subject system with three different density thresholds, which is set by two parameters: the minimum number of variants of a potential *vp* and the maximum number of usage relationships between potential *vp*-s and variants (*i.e.*, usage hops). Namely, the thresholds with (A) $\geq$ 5 variants

Table 3: #nodes identified as being part of dense zones compared to the total number of nodes in all subject systems

Project	*symfinder* nodes	*symfinder-2* ≥ 5 *v*-s ≤ 3 hops	*symfinder-2* ≥ 10 *v*-s ≤ 3 hops	*symfinder-2* ≥ 30 *v*-s ≤ 2 hops
Java AWT	431	28	22	3
Apache CXF	3086	98	32	4
JUnit	118	5	0	0
Maven	616	8	1	0
JFreeChart	578	34	15	3
ArgoUML	1258	40	15	3
Cucumber	331	4	0	0
Logbook	117	0	0	0
Riptide	89	0	0	0
NetBeans	3494	58	22	2

and ≤ 3 usage hops, (B) ≥ 10 variants and ≤ 3 usage hops, and (C) ≥ 30 variants and ≤ 2 usage hops. We have carefully chosen these parameters values, based on a previous empirical evaluation with the ArgoUML, JFreeChart, and Java AWT, and for which we have the best knowledge to manually evaluate the impact of the density threshold. By increasing the threshold on the number of variants, we aim to consider only highly-dense *vp*-s , whereas by decreasing the threshold on the usage hops between such *vp*-s, we aim to consider only highly-dense *vp*-s which are close in terms of usage relationships. The obtained results are given in Table 3.

It can be observed that, in all subject systems, fewer nodes are displayed when using any of the three threshold values. For instance, JFreeChart has in total 578 identified *vp*-s with variants at class level. After applying the three threshold values, 34, 15, and 3 *vp*-s and variants (*i.e.*, entry points) remain, respectively. Moreover, the number of remaining *vp*-s decreases as we increase the minimum number of variants for a *vp*, and decreases the number of usage relationships between both of them. However, these values are not adapted to such projects like Logbook or Riptide, for which no *vp*-s remain with these three default thresholds.

These results suggest that determining a set of appropriate values for the parameters when setting the density threshold is highly dependent on the studied project's characteristics. That is, even some large projects in terms of lines of code, such as Maven, can have a considerable number of potential *vp*-s but with few variants. For such reason, setting a high density threshold may filter out most or all potential *vp*-s with variants (*i.e.*, there will be no entry points). Based on our experiments with ten subjects, we conclude that the first threshold value (*i.e.*, ≥ 5 variants and ≤ 3 usage hops) can be used as a good starting point to begin the exploration and comprehension of the identified variability by *symfinder-2*.

5.4 RQ_3: Usefulness of API-based filtering

As is shown in Table 3, the number of identified potential *vp*-s with variants is extensive (*cf.* nodes). For example, the Cucumber framework has 238 potential *vp*-s with 282 variants at class and method levels. Analysing and comprehending the Cucumber's variability from the provided visualization [13], which exhibits 331 nodes, can be really difficult. But, as a testing framework, Cucumber has an API that exposes classes for defining the dependency steps and an object factory for customizing the dependency injections. Therefore, we defined RQ_3 : **is the API information of a given system useful to simplify its identified variability by *symfinder-2*?** To answer it, we ran *symfinder-2* on three subject systems, namely Cucumber, Logbook, and Riptide, while taking into account the code units annotated by developers using the API Guardian library. We then compared the number of nodes that are displayed in their visualizations before and after using their respective API to filter in the related classes. Results are given in Figure 4.

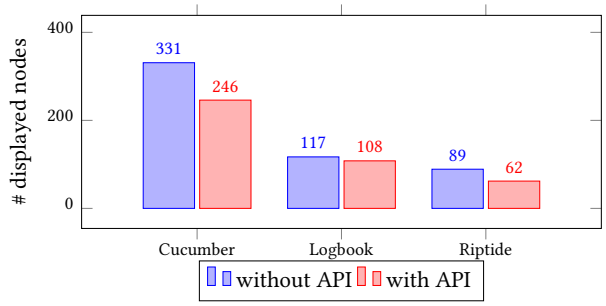

Figure 4: Number of *vp*-s and variants (nodes) displayed on the visualization before and after refinement by API

It can be observed that in all three subjects the visualization with the applied API has notably fewer nodes than the original visualization by *symfinder-2*. We manually checked that, while reducing the number of potential *vp*-s with variants, it always shows those that are considered as the most API relevant. They can help to comprehend each system's variability, that is, to give us an insight on the implemented domain variability. In the three cases, these *vp*-s with variants can be used as entry points for users in order to begin with the variability comprehension of the systems. We interpret this filtering by an API as facilitation in the overview and zooming parts of the Shneiderman mantra [36]: *overview first, zoom and filter, then details on demand*. In the longer term, we believe that this should be extended by the integration of the *symfinder-2* toolchain with other sources that contain variability information for a given system, or simply to show how different variability information sources could be blended together.

5.5 RQ_4: Scalability

Lastly, we defined RQ_4 : **does the identification of usage relationships impact the scalability of *symfinder-2*?** This question aims at determining whether the additional functionalities in *symfinder-2* harm its scalability. The impact could be located in the identification phase, as the usage relationship is identified in the source code representation, as well as in the visualization phase, where new elements are computed to filter all displayed elements. To this end, we measured the time taken to identify and visualize the variability in all ten subject systems. We conducted our experiments on a Linux environment running Arch Linux 5.11.12-arch1-1 x64 with Intel i7-9850H (12 cores) @ 4.6GHz and 32Go memory. The visualizations are tested using Mozilla Firefox 87.0 and Google Chrome 89.0.4389.114.

[13] Cucumber's visualization: https://deathstar3.github.io/symfinder2-demo/visualizations/cucumber-v6.8.0-usage.html

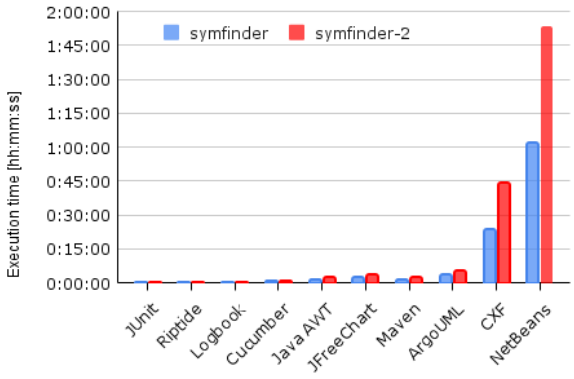

Figure 5: Execution time in *symfinder* and *symfinder-2*

We noticed that the computation needed to render and display the visualization is not very time consuming, with 700 ms on Chrome and 850 ms on Firefox for the NetBeans system. The identification step is clearly the most time-consuming activity. We hence measured and compared this time on the ten subject systems with *symfinder* and *symfinder-2*. Figure 5 summarizes the obtained execution times for both versions (with density calculation based on thresholds of 5 variants and 3 hops). Although the execution time with *symfinder-2* is higher than with *symfinder* for every system, we observe that the difference increases with the size of a system and number of identified *vp*-s and variants, for instance, 20% of difference for Riptide (23 sec → 28 sec) and 85% of difference for NetBeans (01:02:10 → 01:55:04). This can be explained by the fact that a higher number of relationships between classes needs to be parsed and treated by *symfinder-2* in its database. While there seems to have an exponential evolution *w.r.t.* LoC with systems of the size of NetBeans (5 M LoC), we believe the analysis step is still adapted to large systems, as *symfinder* was also successfully applied to Firefox and its 25 M LoC [27]. Then, waiting for around 1 or 2 hours to run *symfinder-2* only on the new releases of a project, for example, every 6 months, is affordable.

5.6 Discussion and threats to validity

Summary of $RQ_1 - RQ_4$. *symfinder-2* provides a more focused identification and visualization of relationships among the potential *vp*-s with variants than *symfinder*. Depending on the system's size, *symfinder-2* can take between 30 seconds to 2 hours to identify between 250 and 11 K potential *vp*-s with variants. Besides, it supports users with up to four ways to begin the variability comprehension of a given system from its visualization. In particular, our experiments suggest using the density threshold with ≥ 5 variants and ≤ 3 usage hops or, if available, the API-based filtering.

Internal threats to validity. A first internal threat concerns the distinction of real *vp*-s and variants from the potential ones proposed by the different versions of *symfinder*. For the considered subject systems (except NetBeans that was too large), we manually determined whether the remained *vp*-s and variants after applying the thresholds represent some variability implementations. We thus did a sample verification by examining identified classes, checking for their documentation on the project website, and devising the kind of variability that was implemented. We could be partially wrong in our interpretation, but we believe it has a limited impact. Then, determining whether an identified *vp* or variant actually implements some domain variability was hard to conclude as non of the subject systems, except ArgoUML, had a ground truth.

External threats to validity. To address the research questions, we used up to 10 subject systems, which vary across domains, size, type, and developers. While the dataset is still small, we have good confidence that the obtained results also apply to other Java-based variability-rich systems of mid-size. Besides, our experiments over NetBeans show that while the toolchain is likely to scale on very large systems, the proposed improvements in *symfinder-2* are not sufficient to comprehend all the implemented variabilities. Entry points and the usage links enable to provide a better overview and better filtering over the system, but it is still difficult to browse effectively towards a comprehension of the system variability. We expect future work on exploiting additional information (*e.g.*, the architecture of the system, component definitions) and on providing a further improved visualization to facilitate this activity.

6 RELATED WORK

Multiple approaches for feature location [20, 25, 32] and feature identification have been proposed [3], but they mainly rely on multiple systems [23, 42] or use a form of feature annotations in their codebase [13, 18, 19]. Unlike them, we use the property of symmetry in object-oriented constructs and design patterns, to identify the implemented variability in a single codebase. Outside the software variability domain, some other works rely on the identification of inheritance and composition to define *hotspots*, being zones of an object-oriented design that are exposed to client software and hence have to be comprehended to ensure reuse [8, 35]. While this identification relies on design patterns detection used to implement reusable interfaces, it was not related to variability implementations.

Previous studies on software API comprehension focus on the extraction of usage patterns relying on unit tests analysis [41], client code analysis [40], or approaches combining client and library code analysis [34], as well as their evolution in time [14, 33] to help developers in library reuse. In the variability domain, APIs have been studied for the variability of their evolution [1], but not to facilitate the comprehension of variability implementations through a visualization approach as in our work.

7 CONCLUSION

As object-oriented variability-rich software systems are often not organized as software product lines, identifying their variability implementations is difficult. While proposing a first solution, the identification method provided by *symfinder* is incomplete as it does not take into account usage relationships between classes and cannot provide appropriate means to help users start the comprehension activity. In *symfinder-2* we extended this method with usage relationships and provided filtering capabilities based on both automatically detected API-related entry points, and a user-defined density threshold. Application to ten systems has shown that *symfinder-2* provides a more focused identification and visualization of object-oriented variability implementations.

In the future, we plan to include in our tooled approach additional object-oriented metrics and usage mechanisms [17] and to exploit architectural information of the systems to improve the comprehension of variability implementations and their evolution over time. We also intend to study other types of visualization that would provide a more intuitive comprehension of the system's implemented variability.

ACKNOWLEDGMENTS

We thank Florian Ainadou, Paul-Marie Djekinnou, and Djotiham Nabagou for their contribution in the development of *symfinder-2*.

REFERENCES

[1] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. 2019. Variability in Library Evolution: An Exploratory Study on Open-Source Java Libraries. In *Software Engineering for Variability Intensive Systems - Foundations and Applications*. Taylor & Francis Group.

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

[3] 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.

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

[5] James O. Coplien and Liping Zhao. 2020. Toward a General Formal Foundation of Design. Symmetry and Broken Symmetry. *(Forthcoming publication)* (2020).

[6] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems (VaMoS'12)*. 173–182.

[7] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95.

[8] Nuno Flores, Diana Soares, Helder Ferreira, and Marco Rodrigues. 2005. HotSpotter: a JavaML-based approach to discover Framework's HotSpots. *Proc. XATA* (2005).

[9] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. 2008. *Head First Design Patterns*. O'Reilly Media, Inc.

[10] Critina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (SSR '01)*. ACM, 109–117.

[11] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2013. Variability in Software Systems — A Systematic Literature Review. *IEEE Transactions on Software Engineering* 40, 3 (2013), 282–306.

[12] Christopher G Healey, Kellogg S Booth, and James T Enns. 1996. High-speed visual estimation using preattentive processing. *ACM Transactions on Computer-Human Interaction (TOCHI)* 3, 2 (1996), 107–135.

[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] Samuel Huppe, Mohamed Aymen Saied, and Houari Sahraoui. 2017. Mining complex temporal API usage patterns: an evolutionary approach. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 274–276.

[15] Ivar Jacobson, Martin Griss, and Patrik Jonsson. 1997. *Software reuse: architecture process and organization for business success*. Vol. 285. acm Press New York.

[16] 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.

[17] Kung-Kiu Lau and Tauseef Rana. 2010. A taxonomy of software composition mechanisms. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 102–110.

[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] Angela Lozano. 2011. An Overview of Techniques for Detecting Software Variability Concepts in Source Code. In *International Conference on Conceptual Modeling (ER '11)*. Springer, 141–150.

[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] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2017. Bottom-up technologies for reuse: automated extractive adoption of software product lines. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 67–70.

[24] Gabriela K Michelon, Lukas Linsbauer, Wesley KG Assunção, Stefan Fischer, and Alexander Egyed. 2021. A Hybrid Feature Location Technique for Re-engineering Single Systems into Software Product Lines. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–9.

[25] Ivan Mistrík, Matthias Galster, and Bruce R Maxim. 2019. *Software Engineering for Variability Intensive Systems: Foundations and Applications*. Auerbach Publications.

[26] Johann Mortara, Philippe Collet, and Xhevahire Tërnava. 2020. Identifying and Mapping Implemented Variabilities in Java and C++ Systems using symfinder. In *Proceedings of the 24th ACM International Systems and Software Product Line Conference-Volume B*. 9–12.

[27] 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.

[28] Johann Mortara, Xhevahire Tërnava, and Philippe Collet. 2020. Mapping Features to Automatically Identified Object-Oriented Variability Implementations: The Case of ArgoUML-SPL. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*.

[29] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media.

[30] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2012. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (2012), 613–637.

[31] Joseph Rosen. 2008. *Symmetry Rules: How Science and Nature are Founded on Symmetry*. Springer Science & Business Media.

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

[33] Mohamed Aymen Saied, Erick Raelijohn, Edouard Batot, Michalis Famelis, and Houari Sahraoui. 2020. Towards assisting developers in API usage by automated recovery of complex temporal patterns. *Information and Software Technology* 119 (2020).

[34] M. A. Saied and H. Sahraoui. 2016. A cooperative approach for combining client-based and library-based API usage pattern mining. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10.

[35] Reinhard Schauer, Sébastien Robitaille, Francois Martel, and Rudolf K Keller. 1999. Hot spot recovery in object-oriented software with inheritance and composition template methods. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 220–229.

[36] Ben Shneiderman. 2003. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization*. Elsevier, 364–371.

[37] 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.

[38] 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 (SPLC '17)*. ACM, 81–88.

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

[40] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.

[41] Zixiao Zhu, Yanzhen Zou, Bing Xie, Yong Jin, Zeqi Lin, and Lu Zhang. 2014. Mining api usage examples from test code. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 301–310.

[42] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. 2012. Feature identification from the source code of product variants. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 417–422.