Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships
Johann Mortara, Xhevahire Tërnava, Philippe Collet, Anne-Marie Dery-Pinna

To cite this version:

HAL Id: hal-03284626
https://hal.science/hal-03284626
Submitted on 12 Jul 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.
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

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

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

1Their 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 JFreeChart2, 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].

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. 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. We follow here the visual principles of perceptive 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.

\[\text{http://www.jfree.org/jfreechart/}\]

\[\text{Figure 1: Inheritance as a local symmetry in JFreeChart}\]

\[\text{Figure 2: Visualization of identified potential vp-s with variants}\]

---

2In this paper, we extend only the support for Java-based systems.

3PiePlot 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.

4It 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.
example, the left part of Figure 2 is denser than the right part. 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.

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-SP1 [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 color 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.

4.1 Handling the usage relationships

To address Issue 1, we extended the identification step by also identifying the usage relationships 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.

---

1 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

2 symfinder-2’s website: https://deathstar3.github.io/symfinder2-demo/

3 Blue names and arrows in Figure 2 have been manually added for illustration.
### 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 *inherence-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.

---

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

11https://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

<table>
<thead>
<tr>
<th>Subject system</th>
<th>Commit</th>
<th>LoC</th>
<th># vp-s</th>
<th># variants</th>
<th>API / Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java AWT</td>
<td>3319fb</td>
<td>69,974</td>
<td>795</td>
<td>1,706</td>
<td>D / L</td>
</tr>
<tr>
<td>Apache CXF</td>
<td>4a7b71</td>
<td>48,655</td>
<td>3,343</td>
<td>7,625</td>
<td>D / F</td>
</tr>
<tr>
<td>JUnit</td>
<td>60aaf6</td>
<td>7,717</td>
<td>109</td>
<td>245</td>
<td>D / F</td>
</tr>
<tr>
<td>Maven</td>
<td>97c98e</td>
<td>105,342</td>
<td>612</td>
<td>1,147</td>
<td>D / A</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>1f6a9f</td>
<td>94,384</td>
<td>926</td>
<td>1,923</td>
<td>D / L</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>d135342</td>
<td>134,367</td>
<td>776</td>
<td>1,959</td>
<td>D / A</td>
</tr>
<tr>
<td>Cucumber</td>
<td>323724</td>
<td>42,662</td>
<td>238</td>
<td>282</td>
<td>A / F</td>
</tr>
<tr>
<td>Logbook</td>
<td>0f06e7</td>
<td>16,210</td>
<td>96</td>
<td>162</td>
<td>A / L</td>
</tr>
<tr>
<td>Riptide</td>
<td>48b03a7</td>
<td>12,626</td>
<td>102</td>
<td>218</td>
<td>A / L</td>
</tr>
<tr>
<td>NetBeans</td>
<td>cade258</td>
<td>5,058,448</td>
<td>3,621</td>
<td>6,736</td>
<td>D / A</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Subject</th>
<th>Nodes</th>
<th>symfinder</th>
<th>symfinder-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java AWT</td>
<td>431</td>
<td>55</td>
<td>2</td>
</tr>
<tr>
<td>Apache CXF</td>
<td>3085</td>
<td>473</td>
<td>1149</td>
</tr>
<tr>
<td>JUnit</td>
<td>118</td>
<td>23</td>
<td>36</td>
</tr>
<tr>
<td>Maven</td>
<td>616</td>
<td>177</td>
<td>172</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>578</td>
<td>54</td>
<td>167</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>1270</td>
<td>123</td>
<td>460</td>
</tr>
<tr>
<td>Cucumber</td>
<td>331</td>
<td>45</td>
<td>122</td>
</tr>
<tr>
<td>Logbook</td>
<td>117</td>
<td>19</td>
<td>40</td>
</tr>
<tr>
<td>Riptide</td>
<td>89</td>
<td>20</td>
<td>37</td>
</tr>
<tr>
<td>NetBeans</td>
<td>3498</td>
<td>504</td>
<td>1666</td>
</tr>
</tbody>
</table>

of symfinder-2. The symfinder-2 and the used data to obtain the presented results are available online in a reproduction package.\footnote{https://zenodo.org/record/4946730}

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\cite{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 \(RQ_1\): \textit{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 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\cite{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 (\textit{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 \(RQ_2\): \textit{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 (\textit{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

<table>
<thead>
<tr>
<th>Project</th>
<th>symfinder</th>
<th>symfinder-2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes ≥ 5 vp-s ≤ 3 hops</td>
<td>≥ 10 vp-s ≤ 3 hops</td>
</tr>
<tr>
<td>Java AWT</td>
<td>431</td>
<td>28</td>
</tr>
<tr>
<td>Apache CXF</td>
<td>3086</td>
<td>98</td>
</tr>
<tr>
<td>JUnit</td>
<td>118</td>
<td>5</td>
</tr>
<tr>
<td>Maven</td>
<td>616</td>
<td>8</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>578</td>
<td>34</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>1258</td>
<td>40</td>
</tr>
<tr>
<td>Cucumber</td>
<td>331</td>
<td>4</td>
</tr>
<tr>
<td>Logbook</td>
<td>117</td>
<td>0</td>
</tr>
<tr>
<td>Riptide</td>
<td>89</td>
<td>0</td>
</tr>
<tr>
<td>NetBeans</td>
<td>3494</td>
<td>58</td>
</tr>
</tbody>
</table>

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 RQ3: 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. Analyzing and comprehending the Cucumber’s variability from the provided visualization 39, 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 RQ3: 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.

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 RQ4: Scalability

Lastly, we defined RQ4: 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.
We observed that the difference increases with the size of a system with symfinder for their documentation on the project website, and devising the did a sample verification by examining identified classes, checking the thresholds represent some variability implementations. We thus vp subject systems (except NetBeans that was too large), we manually symfinder posed by the different versions of vp distinction of real experiments suggest using the density threshold with ≥ hension of a given system from its visualization. In particular, our supports users with up to four ways to begin the variability compre-
tify between 250 and 11 K potential vp-s with variants. Besides, it play the visualization is not very time consuming, with 700 ms to run symfinder-2 or 2 hours to run 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 RQ1 – RQ4, 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 compre-
hension 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 pro-
posed by the different versions of symfinder. For the considered subject systems (except NetBeans that was too large), we manually determined whether the remaining 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.
REFERENCES


