Identifying and Mapping Implemented Variabilities in Java and C++ Systems using *symfinder*

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

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

Xhevahire Tërnava
t.xheva@gmail.com
Paris, France

ABSTRACT

Variability is present in most modern object-oriented software-intensive systems, despite that they commonly do not follow a product line approach. In these systems, variability is implicit and hardly documented as it is implemented by different traditional mechanisms, namely inheritance, overloading, or design patterns. This hampers variability management as automatic identification of variation points (*vp*-s) with variants is very difficult. *symfinder* is a symmetry-based tooled approach that enables automatic identification of potential *vp*-s with variants in such systems. Then, it visualizes them relying on their density in code assets. From the Java-only version presented at SPLC'2019, we present here several notable improvements. They concern an added support for C++ systems, the identification of *vp*-s implemented by Decorator and Template pattern instances, an enhanced visualization (*e.g.*, to display all variants, and package coloring), as well as automation of the mapping of potential *vp*-s to domain features.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Object oriented development*; **Reusability**.

KEYWORDS

Identifying software variability, visualizing software variability, object-oriented variability-rich systems, tool support for understanding software variability, software product line engineering

1 INTRODUCTION

Most modern software-intensive systems, ranging from small-scale embedded systems to large-scale enterprise ones, are variability-intensive [7]. Our work focuses on systems where the variability among its software products is implemented in a single code-base using traditional object-oriented techniques, such as inheritance, overloading, or design patterns [4, 13]. Contrary to software product lines [4], where variability can be at least partially managed in relation with domain features [3], in these systems, neither the domain variability nor the implemented variability is explicit or systematically documented, hindering the overall comprehension of the implemented variability, and even more the possibility to map it to domain features. Existing tools aiming at identifying features rely on annotations [2] or use a set of *clone-and-own* systems [10]. However, few systems have their features annotated and they mostly manage their variability internally.

Facing this problem, the aim of the *symfinder* toolchain is to facilitate the identification of potential variation points and variants when diverse object-oriented implementations are used together to realize variability in code. *symfinder* relies on an automatic identification of local symmetries (Section 2) common to all object-oriented variability mechanisms. It also provides a visualization in which classes being potential *vp*-s or variants are represented as nodes in a graph that are linked through their inheritance relationships, while the properties of these nodes are used to represent metrics on the number of symmetries (*e.g.*, number of overloaded methods). We have presented *symfinder* at SPLC'2019 in its first version dedicated to single code bases implemented with the Java language [11], and in this paper we introduce some new features of *symfinder*[1].

The toolchain now supports code bases in C++, with a specific parser that takes into account the whole source code without passing through preprocessor directives. Independently of the language, the automatic identification part now supports two additional design patterns, *i.e.*, Decorator and Template, covering all the main patterns used for variability implementation. In a re-engineering context, annotation-based techniques may be used to map the domain features to the corresponding code assets and extract a product line [6]. Using these feature traces in the source code, *symfinder* can now automatically map the identified potential *vp*-s with variants that are relevant to some given domain features. On the visualization side, *symfinder* also allows to color nodes depending on their package in the code assets, and software developers can show or hide variants to facilitate browsing in the graph.

2 BACKGROUND

The different mechanisms used to implement variability in a variability-rich object-oriented system, namely inheritance, overloading, and design patterns, are mainly used to provide a better, and ideally more reusable, design. All elements of such a design have a common property, parts of it change while another part remains unchanged, corresponding to the general definition of symmetry. Symmetry or more particularly local symmetry is encountered everywhere in nature and, according to Alexander's theory of centers, is used to make order in human-made artifacts [1]. In software, whenever object-oriented mechanisms are used to implement some variability, they also exhibit the property of symmetry [5, 14, 15]. This arises since the commonality and variability addressed by such a mechanism also represent the unchangeable and changeable parts in code assets, respectively. For instance, inheritance allows us to factorize common parts of multiple classes into a superclass. The superclass accommodates the commonality, that is, the unchangeable part, of its subclasses, which are the changeable parts and hence accommodate the variability. These variability mechanisms also exist at the method level, such as when overloading methods or constructors

[1] *symfinder*'s artifacts (visualizations, source code, guidelines, demonstration video) are available online at https://deathstar3.github.io/symfinder-demo/splc2020.html.

Table 1: Seven mechanisms, their symmetries, and their respective visualization as nodes with their relationships

Mechanism	Visual	Commonality	Variability	Mechanism	Visual	Commonality	Variability
Class as type	🔴	Class	Objects	Constructor overloading	🔴	Structure	Signatures
Template (in C++)	🔴	Template class	Instances & specializations	Factory Pattern	Ⓕ	Abstract Creator & product	Concrete creators & products
Class subtyping	✹	Superclass	Subclasses	Strategy Pattern	Ⓢ	Strategy interface	Algorithms
Interface	●	Type	Implementation classes	Decorator Pattern	Ⓓ	Components & decorator interfaces	Concrete components & decorators
Method overloading	🔴	Structure	Signatures	Template Pattern	Ⓣ	Method template	Method steps
Inheritance	→						

are used. Then, design patterns rely on both class and method level mechanisms to create more complex designs involving reuse.

The unchanged and changed parts in design are commonly abstracted in terms of variation points (*vp*-s) with variants [13], and thus represent places in code assets with a local symmetry (see [14] for more details). Based on this approach, we built *symfinder* to automatically identify and visualize the variability places of a Java, and now also C++, variability-rich system. It can identify and visualize symmetry in seven common variability mechanisms, summarized in Table 1, including their unchangeable and changeable parts and how they are visualized in our graph-based representation, which is also used in other approaches [8]. Figure 1 depicts the main elements of the toolchain, from source fetching on several git repositories, symmetry identification in a Neo4j graph database, and to the generation of a D3.js based visualization. The whole toolchain is itself dockerized to facilitate its usage.

3 EXTENDED IDENTIFICATION SUPPORT

In order to broaden the scope of variability-rich systems that can be analysed by *symfinder*, we extended the analysis front-end to support C++ (*cf.* Figure 1) and to detect more design patterns.

We first developed a second parser using the ANTLR parser generator and a grammar supporting C++14. Software written in C++ is known to make use of CPP macros, which are likely to implement variability as well. As a first step, we decided to handle these macros to be able to identify symmetries in all C++ code assets, but without representing the variants potentially created by the preprocessor directives. While this is obviously an interesting feature, and part of our future work, we decided to focus first on the symmetry based approach. Specifically, during a first analysis of the C++ code assets in ③', macro definitions are extracted in a file and used to precompile the system using the C preprocessor and to expand macros. Similar to the Java version, a second analysis in ③ generates an AST, which is used to build a graph representation of the source code, stored in a graph database. It is then queried to identify symmetries in OO variability implementation mechanisms.

The identification approach is also similar to the Java one with few adaptations for the mechanisms, such as between the interfaces in Java and virtual pure methods in C++. We added the identification of *vp*-s implemented by the template mechanism in C++, with both their instantiation and specialization identified as variants. We also included the identification of *vp*-s with variants implemented by two more design patterns, namely Decorator and Template patterns, in both Java and C++ systems (*cf.* Table 1).

4 AUTOMATED MAPPING OF VARIABILITIES

With variability implementations identified in code assets, one valuable scenario is to find correspondences between domain features and the *vp*-s with variants. In a previous work, we reported on an experiment with the application of *symfinder* to the ArgoUML-SPL [6]. This system contains a ground truth for domain features with their traces in code assets [9]. By using this ground truth, we manually mapped the potential *vp*-s with variants to domain features in ArgoUML, found out those that are actual *vp*-s with variants, and then calculated precision and recall for the *symfinder* identification [12] (definitions are given below). In complement to the mapping experiment manually made in [12], the presented version of *symfinder* now automates this mapping process and provides an enhanced visualization.

The automated mapper is a set of Python scripts deployed in a separate Docker container. In the *symfinder* configuration file, one can simply provide a `traces` property that specifies the directory where features traces are located, following a well-defined format specification available on *symfinder*'s website [2]. After *symfinder*'s execution, these traces are used together with the JSON file containing the potential *vp*-s and variants output by the *symfinder* engine to build the mapping [3]. Two measures are also calculated: *precision* (*i.e.*, the proportion of potential *vp*-s and variants that have a mapping to features) and *recall* (*i.e.*, the proportion of features traces used for the mapping). As output, a new JSON file containing potential *vp*-s with variants and their traces to domain features is produced. This file is used to obtain a visual representation of the mapping, as shown in Figure 2.

This visual representation aims at facilitating manual feature location activities, as feature traces, when available, are important information that bring domain knowledge in the code assets, but imply manual code inspection and file browsing. Hence we added in the visualization of potential *vp*-s with variants the option to display the feature traces that are mapped. For example, in the visualization of the ArgoUML experiment, one can access information for both domain and implementation variabilities (*cf.* Figure 2). Nodes on the visualization represent classes and interfaces linked

[2]https://deathstar3.github.io/symfinder-demo/splc2020.html
[3]For the sake of simplicity, this part of the toolchain is not shown in Figure 1.

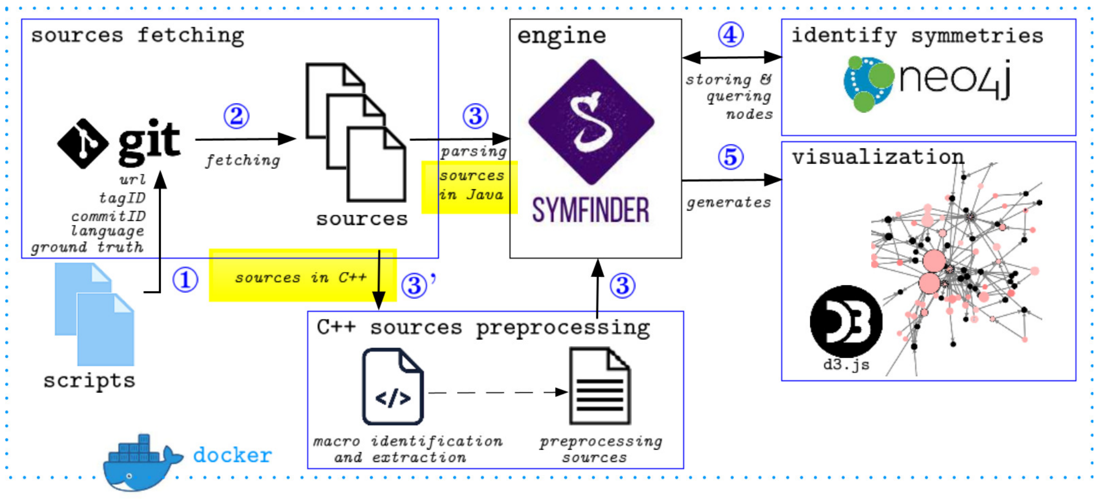

Figure 1: The dockerized *symfinder* toolchain for Java and C++ systems

through inheritance relationships. A node with a blue border indicates that a mapping to domain features exists, hence that the class or interface is an actual *vp* or variant.

5 VISUALIZATION IMPROVEMENTS

In addition to the visualization of used mechanisms to implement potential *vp*-s with variants and their total number, we added two more options in the visualization: *(i)* to color potential *vp*-s with variants within a specific package, and *(ii)* to visualize all variants.

During the application of *symfinder* in real systems [14], we observed that usually those potential *vp*-s with variants that originate from the same package are also part of the same visualized tree. Therefore, to further support the comprehension of implemented variability in a system, we added the option for coloring potential *vp*-s with variants within a specified package. Hence, the `'Color packages'` button on the top bar of the visualization opens a menu where the user can input a package name, namespace (in C++), or class name, whose potential *vp*-s with variants will be colored. The user can color multiple packages and/or classes, and for each one

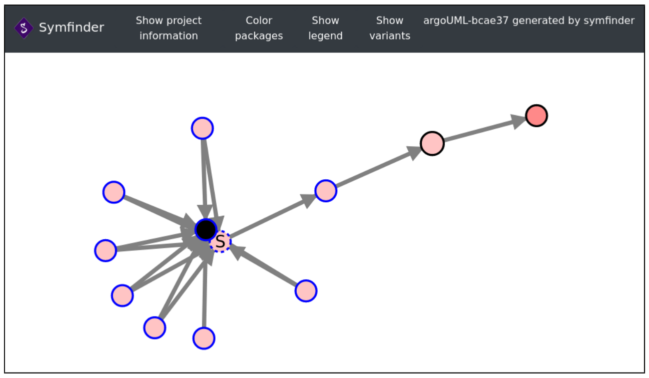

Figure 2: An excerpt from ArgoUML. *Nodes with a blue border are mapped to features, their feature traces are displayed when hovering the node.*

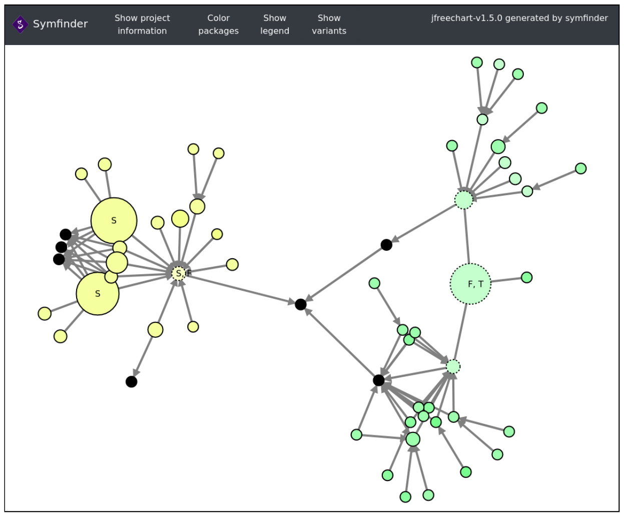

Figure 3: Some colored packages from JFreeChart. `org.jfree.chart.plot` *in yellow and* `org.jfree.chart.renderer` *in green.*

symfinder will automatically generate a new color. An example of visualization with colored packages is given in Figure 3, showing a good separation of concerns between packages in that case.

The first visualization by *symfinder* displayed all potential *vp*-s with class granularity and only their variants that have *vp*-s with a method granularity. However, during the measurement of *symfinder*'s precision and recall [12], we noticed that a considerable number of feature traces were mapped to class level variants, which were hidden in the visualization. Because of their importance, we added the option to visualize at once all class level variants, including those that are without method level *vp*-s. Still, we left available the option to also hide them, as on large systems visualizing all variants considerably overloads the visualization, although nodes

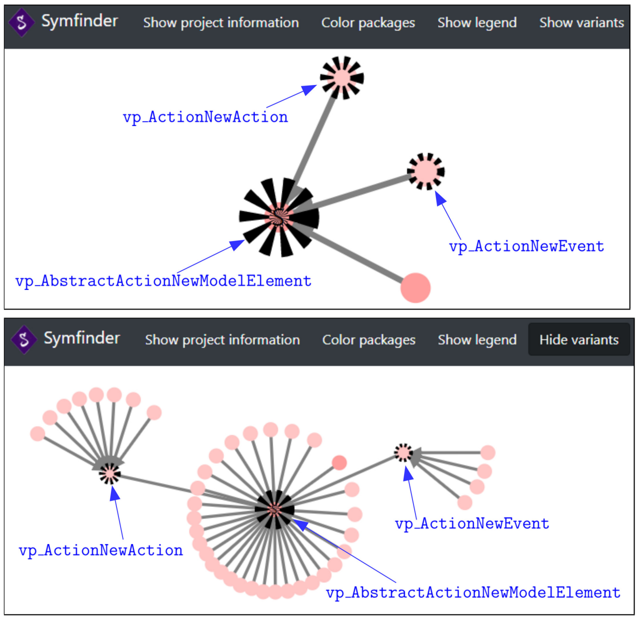

Figure 4: *vp*-s without (top) and with variants (bottom).

can be moved to isolate trees. For example, ArgoUML's visualization exhibits 539 nodes when visualizing potential *vp*-s without variants and up to 1 233 nodes when all their variants are visualized. Taken from ArgoUML's visualization, Figure 4 illustrates the case when *vp*-s with class granularity are visualized without variants (top) and with variants (bottom.) For this reason, we provided a toggle button on the visualization from where all class level variants can be visualized or not. By default, only potential *vp*-s with their variants that have method level *vp*-s are visualized.

6 CONCLUSION

symfinder is a symmetry-based toolchain that enables automatic identification and visualization of potential variation points with variants in object-oriented variability-rich systems organized in a single code base. The toolchain has been improved on its identification engine part, which is now able to analyse Java and C^{++} systems, to identify potential *vp*-s with variants implemented with more design patterns, and on its visualization to facilitate variability comprehension. Automated mapping of variabilities is also supported to relate domain features to potential implemented *vp*-s and variants using pre-existing feature traces.

In the future, we intend to improve the toolchain by providing a direct access to the visualized code assets to further assist the user. We also aim to further validate the approach, with its new additions, in other real and large Java and C^{++} systems. We also plan to report on an experiment of using *symfinder* by the software architect of a real variability-rich system. In this way, we expect to discover the need for other improvements and enhancements.

ACKNOWLEDGMENTS

We thank Théo Foray, Grégoire Peltier and Nathan Strobbe for their contribution in the development of the C^{++} support in *symfinder*.

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] Berima Andam, Andreas Burger, Thorsten Berger, and Michel RV Chaudron. 2017. FLOrIDA: Feature Location Dashboard for Extracting and Visualizing Feature Traces. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '17).* ACM, 100–107. https://doi.org/10.1145/3023956.3023967

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines.* Springer.

[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. 2000. Symmetry Breaking in Software Patterns. In *International Symposium on Generative and Component-Based Software Engineering (GCSE 2000).* Springer, Springer, 37–54.

[6] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. 2011. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *2011 15th European Conference on Software Maintenance and Reengineering.* IEEE, 191–200. https://doi.org/10.1109/CSMR.2011.25

[7] Matthias Galster. 2019. Variability-Intensive Software Systems: Product Lines and Beyond. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '19).* ACM, 1–1. https://doi.org/10.1145/3302333.3302336

[8] 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. https://doi.org/10.1002/smr.1912

[9] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18).* ACM, 257–263. https://doi.org/10.1145/3233027.3236402

[10] 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 *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C '17).* IEEE, 67–70. https://doi.org/10.1109/ICSE-C.2017.15

[11] 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 (SPLC '19, Tools and Demonstrations).* ACM, 5–8. https://doi.org/10.1145/3307630.3342394

[12] Johann Mortara, Xhevahire Tërnava, and Philippe Collet. 2020. Mapping Features to Automatically Identified Object-Oriented Variability Implementations-The Case of ArgoUML-SPL. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20).* ACM, 1–9. https://doi.org/10.1145/3377024.3377037

[13] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. 2005. A Taxonomy of Variability Realization Techniques. *Software: Practice and experience* 35, 8 (2005), 705–754. https://doi.org/10.1002/spe.652

[14] 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.* 231–243. https://doi.org/10.1145/3336294.3336311

[15] Liping Zhao. 2008. Patterns, Symmetry, and Symmetry Breaking. *Commun. ACM* 51, 3 (2008), 40–46. https://doi.org/10.1145/1325555.1325564