symfinder: A Toolchain for the Identification and Visualization of Object-Oriented Variability Implementations

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

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

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

ABSTRACT

When variability is implemented into a single variability-rich system with object-oriented techniques (*e.g.*, inheritance, overloading, design patterns), the variation points and variants usually do not align with the domain features. It is then very hard and time consuming to manually identify these variation points to manage variability at the implementation level. *symfinder* is a toolchain to automatically identify and visualize these variability implementation locations inside a single object-oriented code base. For the identification part, it relies on the notion of symmetry between classes or methods to characterize uniformly some implementation techniques such as inheritance, overloading, or design patterns like Factory. The toolchain also generates an interactive Web-based visualization in which classes that are variation points are nodes linked together through their inheritance relationships, while the size, color, and texture of the nodes are used to represent some metrics on the number of overloaded constructors or methods. As a result, the visualization enables one to discern zones of interest where variation points are strongly present and to get relevant information over concerned classes. The toolchain, publicly available with its source code and an online demo, has been applied to several large open source projects.

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

At the domain level of a Software Product Line (SPL), variability of products is usually captured through common and variable domain features, these features being realized in software assets, such as code, at the implementation level. From the many existing variability implementation techniques, annotation-based ones, such as condition compilation with preprocessors [12], have been heavily used in embedded systems, but they tend to pollute code with a low abstraction level [11] and to produce errors at derivation time [9]. Furthermore, feature modules is a specific technique that structures all lines of code corresponding to a domain feature into a coding unit called feature [2]. While it looks preferable, it does not support well cross-cutting variability [8], and it implies code refactoring that may be completely unfeasible in practice for many systems. This is typically the case in variability-rich systems that have progressively introduced variability into object-oriented code, using many different traditional techniques, such as inheritance, overloading, and design patterns [4, 17]. Variability implementations then do not align well with domain features and identifying where these implementations are precisely located is crucial to manage this kind of variability [15].

While approaches and techniques have been proposed to partially locate domain features at the code level [3, 16], there is no work dealing with the identification of object-oriented variability implementations at the structural level, namely at the level of variation points (*vp*-s) and variants [14, 17]. Contrary to a feature related to the variability domain, a variation point represents one or more locations in code at which variation will occur, while the way that a variation point is going to vary is defined by its variants [7].

This paper introduces *symfinder*, a toolchain to automatically identify and visualize these variability implementation locations inside the code base of a single variability-rich Java system. For the identification part, it relies on the notion of symmetry (defined in Section 2) [5, 20] between classes or methods to characterize uniformly variation points and variants from several implementation techniques, such as inheritance, overloading, or design patterns. Identified variation points and variants are stored into a graph database and reused to generate an interactive web-based visualization. This visualization enables one to discern zones of interest where variation points are strongly present and to get relevant information over concerned classes. Details about the usage of symmetry, the identification approach, and its validation over a set of large projects are available in [18].

External information about the *symfinder* toolchain to be demonstrated are as follows:

- *symfinder* source code is publicly available at https://github.com/DeathStar3/symfinder, including some usage guidelines;
- a video demonstrating the visualization part is available at https://www.youtube.com/watch?v=wb3U6MJ7nAM;
- experimental results from [18] and an online demo are available at https://deathstar3.github.io/symfinder-demo/.

In the remainder of this paper, we first give some background on variation points and symmetry in object-oriented code (Section 2). Next, we describe *symfinder*, giving details on the identification and visualization parts, as well as on the portability and performance of the toolchain (Section 3). We summarize current applications (Section 4) and finally, we conclude the paper by briefly discussing future work (Section 5).

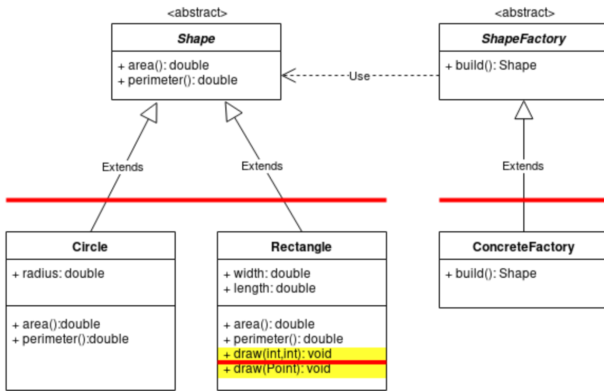

Figure 1: Example of variation points, as local symmetries (illustrated with red lines), in code asset representations

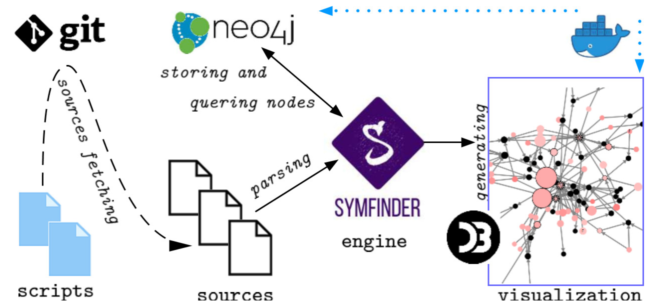

Figure 2: The dockerized *symfinder* toolchain

2 VARIATION POINTS AND VARIANTS IN OBJECT-ORIENTED SYSTEMS

Let us consider an illustrative example with a Java implementation of a family of geometric shapes, such as rectangles and circles (*cf.* Figure 1). What is common from `Rectangle` and `Circle` is factorized into the abstract class `Shape` using inheritance as a variability implementation technique. Besides, overloading is used to implement two ways for drawing shapes, namely the `draw()` method in `Rectangle`. Finally, the `ShapeFactory` and `ConcreteFactory` classes implement a Factory pattern, with an abstract `build()` method that is implemented in the subclass by returning an instance of a subclass of `Shape` depending on some configuration values.

As stated in the introduction, a variation point corresponds to a place in code where some variation happens, and the way this point vary corresponds to variants [7]. Therefore, the example in Figure 1 exhibits three variation points:

- the abstract class `Shape` is common, thus a variation point, for two variants `Rectangle` and `Circle`;
- the abstract creator class `ShapeFactory` is another variation point for its single variant `ConcreteFactory`;
- the method `draw()` in class `Rectangle` is another variation point for two overloaded variants that have different arity.

Symmetry and local symmetry have been recognized as a way to comprehend and create order in nature and human made artifacts [1]. They have also been studied in software and been identified in different constructs of programming languages or object-oriented design patterns [5, 19] where, in accordance with the symmetry definition, a part of their design remains *unchanged* while another may *change*. For example, subtyping in the `Shape` hierarchy in Figure 1 exhibits the property of symmetry. The classes of this type path, `Circle` and `Rectangle`, change while they preserve and conform to the common behavior defined in the superclass `Shape`.

A more complete study of the symmetry in object-oriented constructs can be found in the companion research paper [18], as well as the definition of the relationship between variation points and symmetry. Basically, as variation points and variants, respectively, mark the unchanged and changeable parts in a design, they are actually the local symmetric places in design. Hence, all the techniques used to implement the three variation points in Figure 1 uniformly exhibit the property of symmetry.

Table 1: Six language features, their symmetries, and their respective visualization as nodes with their relationships

Language feature	Visual node	Commonality /Unchange	Variability /Change
Class as type		Class	Objects
Class subtyping		Superclass	Subclasses
Interface		Type	Implem. classes
Method overloading		Structure	Signatures
Constructor overloading		Structure	Signatures
Strategy Pattern	S	Strategy interface	Algorithms
Factory Pattern	F	Abstract Creator and product	Concrete creators and products
Inheritance			

3 SYMFINDER

3.1 Overview

The aim of *symfinder* is to enable automatic identification and visualization of different local symmetries (*i.e.*, variation points) in a variability-rich Java-based system. The main purpose is to provide help in understanding the variability places of a variability-rich system during its development. Figure 2 depicts its whole toolchain, which consists of three parts. First, the sources of a targeted Java project are fetched from its *git* repository, then the *symfinder* engine enables the automatic identification of all its *vp*-s, through the property of local symmetries, and builds a graph representation of them, and finally a visualization of the identified *vp*-s is generated and can be navigated through a web browser.

3.2 Identification through local symmetries

For a targeted variability-rich system, local symmetries are identified according to the defined symmetry in each language construct, technique, and design pattern given in the companion research paper [18] and summarized in Table 1. Specifically, each interface, abstract class, extended class, overloaded constructor, and overloaded method is identified. All together, they actually represent

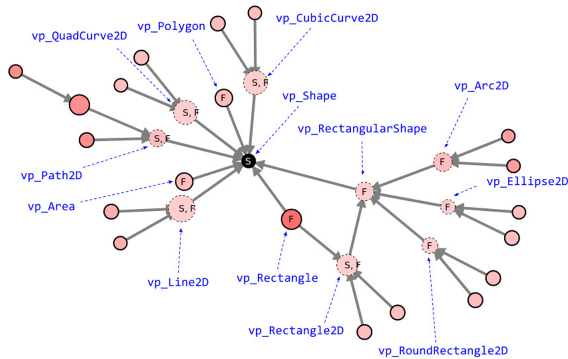

Figure 3: Excerpt of a visualization of identified *vp*-s in the Java AWT library. Annotations in blue show potential *vp*-s names that are displayed when hovering a node.

the potential *vp*-s. Then, the classes that implement or extend them, including the concrete overloaded constructors and methods are also identified, representing their respective variants.

Technically, the *vp*-s identification process is achieved in two steps. First, the targeted Java system is parsed and the structure of its implementation units is stored into a Neo4j graph database where each class, interface, and method is represented by a node, including its structural relationships with other nodes. The node and relationship types are also labeled, for example, `CLASS` and/or `ABSTRACT` for nodes, `EXTENDS` or `IMPLEMENTS` for inheritance relationships. Secondly, *vp*-s are identified by querying the database for specific paths in the labeled graph using the Cypher language [1]. For example, the following query uses labels (*e.g.*, `CLASS`, `EXTENDS`, and `IMPLEMENTS`) to identify some *vp* and its variants:

```
MATCH (c)-[:EXTENDS|:IMPLEMENTS]->(c2:CLASS)
WHERE ID(c) = $id
RETURN count(c2)
```

When the `$id` of the node corresponds to the `Shape` class (*cf.* Figure 1), then the query will count the number of its `EXTENDS` relationships (*i.e.*, 2). Thus, `Shape` is identified as a *vp* with its two variants. Similarly, other queries are used to identify method level *vp*-s, as well as the strategy and factory design patterns. In that last case, more complex queries analyse respectively the structural relationship of classes, and the return types of methods, a detected factory being a class that contains a method returning an object whose type is a subtype of the declared method return type. Moreover, the four main queries used to count the number of *vp*-s and variants, at class and method level are documented here: https://github.com/DeathStar3/symfinder/blob/splc2019-artifact/detection_method.md.

3.3 Web-based visualization

The *symfinder* toolchain provides the capability to generate an interactive Web-based visualization of the identified *vp*-s (*cf.* Figure 2). After considering the visualization capabilities of Neo4j and other visualization forms used in SPL engineering [13], we decided to use the D3.js [2] library as the visualization support, so that only a Web browser is needed to visualize the *vp*-s graph.

Instead of visualizing the graph of *vp*-s with variants by plain nodes and edges, we consider that it is important to also visualize information regarding the used language constructs, techniques, or design patterns for implementing variability. For this reason, as in many software and code artifacts visualizations [10], we rely on the visual principles of preattentive perception [6] using some of the seven parameters that can vary in visualization in order to represent data, namely position, size, shape, value (lightness), color hue, orientation, and texture. The seven kinds of nodes that we use in *symfinder* for the visualization of the kinds of potential *vp*-s with variants are shown in Table 1.

As an example, Figure 3 shows a visualization excerpt of the identified *vp*-s in the Java AWT library. It shows 30 identified *vp*-s without variants, where each *vp* node is represented by a circle, and edges are class extension or interface implementation. Using Table 1, one can interpret that the black circle with letter `S` of `vp_Shape` denotes its relation to an interface that is also a Strategy pattern in code. In addition, the larger size of node `vp_Line2D` denotes that it contains variability at method level through overloading. Similarly, the more intense color of `vp_Rectangle` denotes that it contains variability at constructor level.

The visualization itself has five main features: (1) the visualized graph of *vp*-s can be zoomed in and out, (2) the *vp* label appears when hovering the node, (3) the out-of-scope *vp*-s [18] can be filtered out (*e.g.*, `java.awt.images` when analysing the variability of Java AWT library), (4) the isolated nodes can also be filtered out, and (5) the number of identified *vp*-s and variants.

3.4 Portability and performance

In order to achieve maximum portability, the execution of the *symfinder* toolchain is fully dockerized. Thus, only Docker [3] and Docker Compose [4] are required to run the toolchain. Internally, three Docker environments are executed sequentially, which correspond to the three parts of the toolchain enumerated in Section 3.1:

(1) A Docker container that fetches sources and checks out the desired tags or commits of a variability-rich system from its *git* repository (*cf.* Figure 2). This enables *symfinder* to work easily over any Java system that is publicly available.
(2) The identification process is run by deploying a Docker Compose environment per project that is made of two Docker containers, the *symfinder* engine and an instance of a Neo4j database. Then, a *"runner"* Docker container automates the execution of *symfinder* on multiple systems.
(3) The visualization is provided by running a Docker container that runs a lightweight Python server exposing the generated HTML files on http://localhost:8181.

Finally, the deployment of the toolchain is made through *shell* scripts that encapsulate the Docker commands to run. *symfinder*'s deployment is validated on three operating systems, GNU/Linux,

[1]https://neo4j.com/developer/cypher/

[2]https://d3js.org/

[3]https://www.docker.com/

[4]https://docs.docker.com/compose/overview/

Table 2: Execution time of all steps when analysing JFreeChart 1.5.0.

Step	Execution time [hh:mm:ss]
Detection of methods and classes	00:19:42.702
Creation of inheritance relationships	00:02:24.419
Detection of strategy patterns	00:01:22.376
Detection of factory patterns	00:02:19.115

macOS Sierra 10.12 or newer on hardware from at least 2010, and Windows 10 64-bit (Pro, Enterprise or Education)[5].

Regarding its performance, a *symfinder* execution mainly depends on the size of the analysed system. For example, the execution time over the JFreeChart 1.5.0 (*cf.* Section 4) with 94,384 LoC takes around 26 minutes on a virtual machine with 1 core of Xeon E5-2637 at 3.50GHz and 128 GB of memory, to detect an overall number of 1415 variation points. Details on the time spent on each main step are given in Table 2. The detection of methods and classes is taking 75% of the time as it needs to traverse all classes and methods, while the other steps reuse the built graph.

4 CURRENT APPLICATIONS

We evaluated the identification and visualization of *vp*-s, including the portability and performance of *symfinder*, by conducting several experiments with realistic variability-rich systems. Specifically, we applied *symfinder* over eight open source systems, namely Java AWT, Apache CXF 3.2.7, JUnit 4.12, Apache Maven 3.6.0, JHipster 2.0.28, JFreeChart 1.5.0, JavaGeom, and ArgoUML. Their analysed tags, commits, size in LoC, and some basic metrics on the identified number of *vp*-s with variants are given in the companion research paper [18]. As the main result of these successful applications of *symfinder*, we were able to identify the first patterns of variability in object-oriented variability-rich systems. Still, all conducted experiments are available at https://deathstar3.github.io/symfinder-demo/, which are illustrated with extracted screenshots, explanations, and then a demonstration of their visualization is also deployed online. A specific set of experiments is also available as an online demo.

5 CONCLUSION

symfinder is a toolchain that supports identification and visualization of different kinds of variation points with variants of a variability-rich Java system using the property of symmetry in object-oriented software constructs. The toolchain source code is publicly available and can be easily used through a containerized version on Docker.

Future work includes toolchain extensions to identify symmetries in other language features, being object-oriented or functional, integrating other properties from Alexander's theory of centers [1], and studying how to handle code evolution w.r.t. variation points identification and visualization.

[5]Its deployment is not possible on Windows Home, which is a limitation of Docker.

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] 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, and Kyo-Chul Kang. 2013. *Systems and Software Variability Management.* Springer.

[5] James O Coplien and Liping Zhao. 2000. Symmetry breaking in software patterns. In *International Symposium on Generative and Component-Based Software Engineering.* Springer, 37–54.

[6] Stephan Diehl. 2007. *Software visualization: visualizing the structure, behaviour, and evolution of software.* Springer Science & Business Media.

[7] Ivar Jacobson, Martin Griss, and Patrik Jonsson. 1997. *Software reuse: architecture, process and organization for business success.* ACM Press/Addison-Wesley Publishing Co.

[8] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A variability-aware module system. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 773–792.

[9] Maren Krone and Gregor Snelting. 1994. On the inference of configuration structures from source code. In *Proceedings of 16th International Conference on Software Engineering.* IEEE, 49–57.

[10] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. 2005. Codecrawler: an information visualization tool for program comprehension. In *Proceedings of the 27th international conference on Software engineering.* ACM, 672–673.

[11] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. # ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, 143–150.

[12] 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.

[13] 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.

[14] Angela Lozano. 2011. An overview of techniques for detecting software variability concepts in source code. In *International Conference on Conceptual Modeling.* Springer, 141–150.

[15] Andreas Metzger and Klaus Pohl. 2014. Software product line engineering and variability management: achievements and challenges. In *Proceedings of the on Future of Software Engineering.* ACM, 70–84.

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

[17] Xhevahire Tërnava and Philippe Collet. 2017. On the Diversity of Capturing Variability at the Implementation Level. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B.* ACM, 81–88.

[18] 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.

[19] Liping Zhao. 2008. Patterns, symmetry, and symmetry breaking. *Commun. ACM* 51, 3 (2008), 40–46.

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