Identification and visualization of variability implementations in object-oriented variability-rich systems: a symmetry-based approach

Xhevahire Tërnava, Johann Mortara, Philippe Collet, Daniel Le Berre

Automated Software Engineering manuscript No.
(will be inserted by the editor)

Identification and visualization of variability implementations in object-oriented variability-rich systems: a symmetry-based approach

Xhevahire Tërnava · Johann Mortara · Philippe Collet · Daniel Le Berre

Received: date / Accepted: date

Abstract Most modern object-oriented software systems are variability-rich, despite that they may not be developed as product lines. Their variability is implemented by several traditional techniques in combination, such as inheritance, overloading, or design patterns. As domain features or variation points with variants are not a by-product of these techniques, variability in code assets of such systems is implicit, and hardly documented, hampering qualities such as understandability and maintainability. In this article, we present an approach for automatic identification and visualization of variability implementation places, that is, variation points with variants, in variability-rich systems. To uniformly identify them, we propose to rely on the existing symmetries in the different software constructs and patterns. We then propose to visualize them according to their density. By means of our realized toolchain implementing the approach, *symfinder*, we report on a threefold evaluation, *(i)* on the identified potential variability in sixteen large open-source systems and *symfinder*'s scalability, *(ii)* on measuring *symfinder*'s precision and robustness when mapping identified variability to domain features, and *(iii)* on its usage by a software architect. Results show that *symfinder* can indeed help in identifying and comprehending the variability of the targeted systems.

Xhevahire Tërnava (corresponding author)
Université de Rennes 1, Inria/IRISA, Rennes, France
E-mail: xhevahire.ternava@irisa.fr

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

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

Daniel Le Berre
Université d'Artois, CNRS, CRIL, Lens, France
E-mail: daniel.leberre@cril.fr

Keywords automatic variability identification · variability visualization · object-oriented variability-rich systems · variability comprehention · variability evolution · software product line engineering

1 Introduction

Most modern software-intensive systems, ranging from small-scale embedded systems to large-scale enterprise systems to ultra-large systems of systems, are variability-intensive [36, 30, 29]. Software variability is commonly understood as the ability of a software system or software artifact to be efficiently extended, changed, customized, or configured for use in a particular context [11]. Beyond this, variability is a key fact of most, if not all, systems [36], regardless of whether they are part of a software product line (SPL) or not, although it is largely studied in the context of product lines [76] and product families [11]. As an anticipated change [73], variability evolves and needs to be managed, being a relevant concern for software engineers of such systems.

In systems that are part of a product line, their variability in different levels of abstraction (*e.g.*, at the domain or implementation levels) is commonly documented and managed in terms of features in a feature model [42, 81, 64].. However, this is barely the case in object-oriented variability-intensive system, often referred to as variability-rich systems, that do not follow a complete product line approach [5]. In an object-oriented system, variability among its software products is implemented in a single code-base using traditional techniques, namely inheritance, parameters, overloading, or design patterns [28, 86, 11]. Leaving the domain variability aside, its implemented variability in code assets is neither explicit nor documented, strongly hindering its management. To effectively manage it, software engineers have thus to be aware of where in the code assets is implemented variability.

To be aware of the variability in a variability-rich system, one might choose to extract its variability or to make it explicit by migrating the system to an SPL using any of the reverse or forward engineering approaches. Reverse engineering approaches have notably been used to extract architectural views of existing systems (*e.g.*, [31]), but with the increasing diversity to be managed by software applications, extracting variability information becomes as relevant as other more classical models of existing software. In these approaches, such as feature location [23, 80, 6], feature identification [44, 56], feature delimitation (using a form of annotations) [53], or feature modularization [5], features commonly tend to describe the domain variability of a product line or variability-rich system and are required to be known in advance [64, 77]. But first, domain variability is hardly documented in variability-rich systems [47]. Then, existing reverse engineering approaches use a *set* of software products that are created by a clone-and-own strategy to identify their common and varying features in order to build an SPL [60]. This is actually inapplicable to our considered systems as their varying products share a single code-base. Apart from this, when annotative or modularization approaches are used to

migrate to a product line, despite their respective applicability and advances, they require substantial manual effort or imply a change on top of the underlying design of the system, being it object-oriented or functional.

Considering these reasons, and also that real systems are characterized by a large amount of variability, we expect that in an object-oriented variability-rich system one can (1) keep unchanged its main decomposition of code and still (2) be able to automatically identify variability implementation places in its code assets and (3) use them to comprehend the implemented variability. We consider that the identified variability places can be abstracted in terms of *variation points* (*vp*-s) with *variants*[1] and used to comprehend a system's variability. Hence, to attain these goals which make up the motivation of our work (Section 3), a proper identification and representation of *vp*-s with variants of the targeted systems is needed.

There are studies on how to address variability by traditional techniques [12, 28, 74, 86], but there is a complete lack of approaches to identify *vp*-s with variants [55] implemented with different object-oriented techniques in a single code-base system. This could be due to the fact that each technique differently supports the implementation of *vp*-s with variants [55, 87]. From a reverse perspective, this indicates that depending on the used technique, each *vp* with its variants in code assets requires its own way to be identified.

On the other hand, according to a recent mapping study by Lopez-Herrejon et al. [54], several variability representation approaches are proposed in the context of product lines. Most of them visualize domain variability, that is, features in a feature model. The used terms to conduct this study also include the "variation point" and "variant" terms. However, the results of this mapping study show that while there are few approaches that visualize the variability of code assets, there is a complete lack of those that visualize *vp*-s with variants. This may be a consequence of the previous issue regarding the lack of approaches to identify *vp*-s with variants.

Herein, our contribution is multifold.

- Studying the variability implementation techniques, we observe a conceptual relationship between the exhibit symmetry in traditional object-oriented techniques [16, 100, 99, 34] and *vp*-s with variants as variability abstractions [38]. This leads us to a symmetry-based approach to *uniformly* identify different kinds of *vp*-s with variants, that is, realized by different traditional techniques, within the same variability-rich system (Section 4).
- To show the feasibility of our approach, we developed *symfinder* (Section 5). It automates the identification of *vp*-s with variants, using the revealed *symmetry* in their used techniques, and visualizes them relying on their *density*. *symfinder* currently supports the identification of those *vp*-s with variants that have been realized with two widely used object-oriented language features and four software design patterns.
- We evaluate our tooled approach in sixteen open-source systems, conducting several experiments regarding three research questions, and report their

[1] Their definition is given in Section 2.1

results (Section 6). First, using all subjects, we report on the amount of identified variability in real variability-rich systems and the *symfinder*'s scalability (Section 7). Secondly, using two subjects, we evaluate the precision and robustness of our approach, grounded on symmetry in traditional techniques, to identify *vp*-s with variants (Section 8). Thirdly, we report an experience of a software architect on the use of *symfinder* in his own system (Section 9). The last two evaluations also show that the understandability of a variability-rich system is then improved by making explicit its expected variability to its software architects.
– Lastly, *symfinder* is publicly available [2], as well as all our conducted experiments [3]. A stable version of them is archived in `https://doi.org/10.5281/zenodo.5872420`.

While threats to validity (Section 10), and related work (Section 11) are discussed, we conclude the paper by evoking future work (Section 12).

An earlier version of this approach appeared in a conference [89, 68]. There, we reported the initial idea of the symmetry-based approach and its realization in *symfinder*. Here, as a follow-up to it, both of them are substantially extended. We reveal symmetry in two additional software design patterns and extend *symfinder* accordingly, including a better rendering of variants in the visualization. Moreover, the evaluation part is at the same time new and wider in its approach. For the first kind of evaluation, we double the number of subjects and made more precise observations. As a new evaluation, we measure the precision and robustness of our tooled approach by substantially extending the case study presented in [70] and providing a new one. We finally provide an experience report on the usage of *symfinder* by a software architect.

2 Background

In this section, we provide the basic concepts of variation points with variants in code assets of an object-oriented system, as well as the revealed symmetry in object-oriented language constructs and software design patterns.

2.1 Variability in code assets of an object-oriented system

Let us consider an illustrative example of a family of geometric shapes, such as rectangles and circles, with a Java implementation, given in Listing 1. What is common between the `Rectangle` and `Circle` classes is factorized into the abstract class `Shape` using inheritance as a variability implementation technique [38, 12]. Namely, the way for setting a new origin for `Rectangle` and `Circle` is common, which is factorized in lines 3–7 within class `Shape`, while each of them has a different way for calculating the area and perimeter.

[2] `https://github.com/DeathStar3/symfinder`
[3] `https://deathstar3.github.io/symfinder-demo/jrn20.html`

```
1  /* Class level variation point, vp_Shape */
2  public abstract class Shape {
3    private Point origin;   // Point defined
4    // Constructor omitted
5    public void newOrigin(Point o) {
6      origin.setPoint(o.getX(), o.getY());
7    }
8    public abstract double area();
9    public abstract double perimeter(); /*...*/
10 }
```

```
11 /* First variant, v_Rectangle, of vp_Shape  */
12 public class Rectangle extends Shape {
13   private final double width, length;
14   // Constructor omitted
15   public double area() {
16     return width * length;
17   }
18   public double perimeter() {
19     return 2 * (width + length);
20   }
21   /* Method level variation point, vp_Draw  */
22   /* Variant v_drawCoordinates of vp_Draw  */
23   public void draw(int x, int y) {
24   // rectangle at (x, y, width, length)
25     System.out.println("Rectangle at (" + x + ", " + y + ")");
26   }
27   /* Variant v_drawPoint of vp_Draw  */
28   public void draw(Point p) {
29   // rectangle at (p.x, p.y, width, length)
30     System.out.println("Rectangle at (" + p.getX() + ", " + p.getY() + ")"
       ;
31   }
32 }
```

```
33   /* Second variant, v_Circle, of vp_Shape  */
34   public class Circle extends Shape {
35     private final double radius;
36     // Constructor omitted
37     public double area() {
38       return Math.PI * Math.pow(radius, 2);
39     }
40     public double perimeter() {
41       return 2 * Math.PI * radius;
42     }
43   }
```

Listing 1: Example of variability implementations. The `vp_Shape` and `vp_Draw` represent two *vp*-s at the class and method levels, respectively

Besides, overloading is used to implement the two ways for drawing any of the considered shapes, namely the `draw()` method in `Rectangle`, lines 22–26 and 27–31. Despite its small size, we consider this example as representative of code assets in a variability-rich system where several object-based techniques are used together, such as inheritance, overloading, or design patterns, in a single code-base.

When object-oriented or functional programming paradigms are used, code assets consist of three parts: core, commonalities, and variations [91, 14, 7]. The core part is what remains of the system in the absence of any particular

feature, that is, the assets that are included in any of its final software products [91]. A commonality is the common part between the related variations of given code assets, while variations indicate how and when should code assets vary [36]. Variability of code assets is used to differentiate the software products within an SPL or variability-rich system. After commonalities are factorized from the variations and implemented, commonalities can be considered as part of the core [91], except when they represent some optional code assets, in which case they become part of variations [88]. Such commonalities and variations are usually abstracted in terms of variation points (*vp*-s) with *variants*, respectively [38, 20, 77], which are related to concrete elements in code assets [40]. In the presented work, we state the following definition.

Definition 1 A variation point identifies one or more locations at which the variation will occur [38], while the way that a variation point is going to vary is expressed by its variants.

For instance, based on the presented concepts and Definition 1, the class `Shape` in Listing 1 is common, thus a class level variation point, for two variants `Rectangle` and `Circle`. Likewise is the method level variation point of `draw` (lines 21–31) with its two variants in lines 22–26 and 27–31. In this example, variability is implemented by inheritance and overloading, respectively.

2.2 Symmetry in object-oriented software constructs

Most of the object-oriented language constructs and software design patterns, such as classes, inheritance, overloading, overriding, strategy pattern, and decorator pattern, are well-known as traditional variability implementation techniques [38, 5]. Besides, inspired by Alexander's theory of centers [1], several other works show that object-oriented techniques and software design patterns also exhibit a form of symmetry [16, 15, 99, 100, 34, 98].

Definition 2 Symmetry is immunity to a possible change [78, 79].

This is the common symmetry definition, which has two components: (1) *possibility of change* and (2) *immunity*. For instance, in natural sciences, the symmetry of an object is a *transformation* (*e.g.*, reflection, rotation, or translation) that leaves the object seemingly *unchanged* [85]. Whereas, *the density of local symmetries* in an object or structure is crucial for measuring its coherence [2, 1]. Conforming to Definition 2, the two components of symmetry are also observed in object-oriented language constructs and software design patterns [16, 100, 99, 98]. In Figure 1 is illustrated the observed symmetry in inheritance and overloading, using the example in Listing 1. Specifically, in class subtyping all classes of a type path may change, but they must preserve and conform to the common behavior. Thus,

– the *possibility of a change* in the abstract class `Shape` materializes in its potential different subtypes, such as `Rectangle` and `Circle`. Their shown change regards the way the area and perimeter are computed. Whereas,

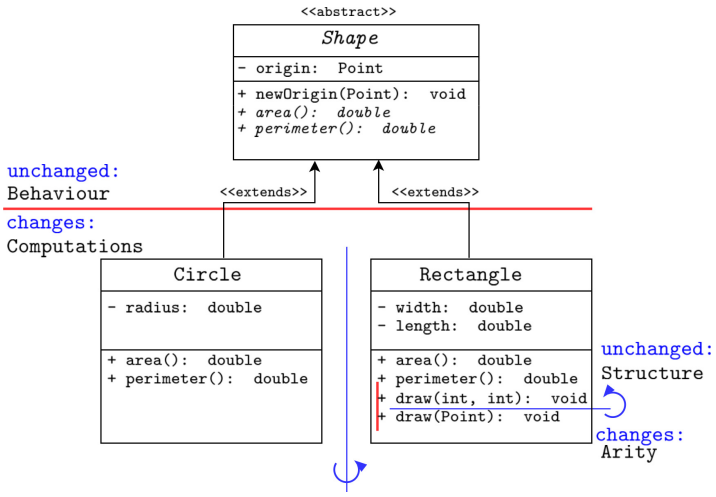


Fig. 1: An illustration of symmetry in inheritance, as subtyping, and overloading using the design view of the given example in Listing 1

– the *immunity to change* maps to these subtypes preserving the behavior of their supertype `Shape`, such as the realized behavior in `newOrigin(Point)`.

Hence, a class subtyping defines a *substitution symmetry* for its subtypes, which can be substituted as they have the same supertype. Class subtyping is only one of the ten well-known forms of inheritance [65, p. 822]. According to a forthcoming study, the other forms of inheritance also exhibit the property of symmetry and can be described similarly [17].

Likewise, symmetry appears also in software constructs at method or function level. For instance, the method or function overloading lets you define multiple functions of the same name but with different implementations. For example, for each overloaded method `draw()` of `Rectangle` in Listing 1 (*cf.* lines 22-26 and 27-31):

– the number of the taken parameters have *changed*, whereas
– the name and the return type have remained the same, *unchanged*.

This denotes the symmetry in overloading, which is also illustrated in Figure 1, where the name of an overloaded function remains unchanged while its arity or types of its parameters change. Thus, overloading also defines a substitution symmetry for the overloaded methods, meaning that they can be substituted from one to another. Both kinds of substitutions, as symmetry transformations, are illustrated by blue lines in Figure 1.

For most of the object-oriented language constructs and software design patterns, it has been shown that under a certain transformation, such as substitution in subtyping or overloading, a specific property of the system is preserved, such as behavior in case of subtyping or structure in case of overloading [98]. This indicates that any of them can be described in terms of

symmetry and appear as *local symmetries* in the wholeness of code assets in an object-oriented software system.

3 Motivation

In many object-oriented variability-rich systems with a single code-base that do not follow a product line approach [76, 5], variability in code assets is implemented by different traditional techniques, such as inheritance, parameters, overloading, or software design patterns [28, 86, 11]. The identification of such variability is essential for its management, such as to maintain, evolve, or potentially map the implemented variability to the system's domain features [63].

But, the code units that structure such systems, namely classes or functions, do not align well with domain features [4, 5]. Then, *vp*-s with variants in code assets are not a by-product of traditional techniques [9]. For the mentioned reasons in Section 1, instead of identifying *domain features* in code assets, identifying the varying implementation elements directly in code assets, that is, *variation points* with their *variants*, seems to be the first and necessary step to comprehend the variability of the class of systems we consider.

To the extent of our knowledge, there is no automated approach for identifying *vp*-s with variants in our context of object-oriented traditional techniques. A possible reason is that traditional techniques are too diverse [55]. There is also a lack of approached to represent *vp*-s with variants in code assets [54]. Hence, identifying and representing *vp*-s with variants that are realized by traditional techniques seems to be cumbersome and a non-trivial activity with poor support.

Actually, the diversity of traditional techniques is analysed in different frameworks, taxonomies, and catalogs, by comparing them on different criteria [86, 74, 28, 26, 5]. For instance, in a recent catalog, 16 traditional techniques are compared and classified based on 24 properties [87]. But, despite these comparative schemas, we were not aware that any common property of these techniques exists, which property could be used to identify different kinds of *vp*-s with their variants in a uniform or automated way. For example, in Listing 1, the *vp* `Shape` has a *class level* granularity and is resolved at *runtime* during product derivation, whereas the *vp* `Draw` has a *method level* granularity and in our case of a Java-based implementation is resolved at *compile time*. Both of them resemble two *vp*-s with four different properties that should be considered during their identification.

Towards a uniform approach to identify *vp*-s with variants, we noticed that the majority of traditional techniques have been shown to be describable in terms of symmetry or local symmetry [16, 99, 100], which is introduced in the previous section. Consequently, we propose an approach based on the property of *(local) symmetry* in traditional techniques to uniformly identify *vp*-s with variants of a given variability-rich system. Then, as a means to easily distinguish zones of interest with its variability, we rely on the *density* of *vp*-s with variants to build a suitable variability visualization support.

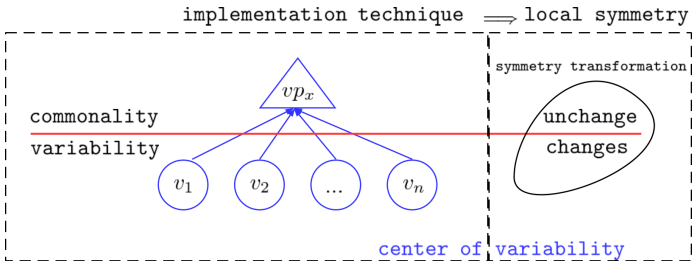


Fig. 2: An illustration of variability concepts and their relationship as given in Definition 3: a variation point (vp_x) with its variants ($v_1, v_2, ..., v_n$), commonality, variability, implementation technique, symmetry transformation, unchange, changes, local symmetry, and center of variability

4 A symmetry-based approach

We now give the main principles of the proposed approach based on symmetry for identifying potential *vp*-s with variants and on the density of these symmetries for comprehending the implemented variability of a system.

4.1 *vp*-s with variants as local symmetries

We argue that the object-oriented language constructs and software design patterns can be seen from two perspectives: (1) as software constructs that are characterized with the property of symmetry and (2) as traditional variability implementation techniques. By combining these two perspectives, we propose an approach for identifying potential variability implementations of a system through pinpointing places with symmetry in its code assets.

Perspective (1): According to [98], the overall symmetry of object-oriented systems organized in classes is usually broken by introducing interfaces, abstract classes, while the rise of software design patterns is also seen as a reaction to this problem [16, 98]. Based on this, we first argue that the usage of any traditional technique for implementing the variability of a system, such as class subtyping, overloading, or design patterns, denotes the existence of a local symmetry in the wholeness of its code assets.

Perspective (2): Then, using the fact that each implementation technique is commonly abstracted in terms of a variation point (*vp*) with its variants (*cf.* Section 2.1), we make the assumption that a *vp* with variants can be identified by the property of local symmetry. Specifically, while *vp*-s resemble the unchanged parts (*i.e.*, commonality) in the design of reusable code assets, variants resemble their changed parts (*i.e.*, variability). The interrelationship between all these concepts, in two perspectives, is illustrated in Figure 2.

Table 1: Eight object-oriented software constructs and their symmetries

Software construct	Commonality / Unchanged	Variability / Change	Symmetry transformation
Class subtyping	Superclass / Type	Subclasses	Substitution
Method overloading	Structure	Signatures / Arity	Substitution
Class as type	Class / Constructor	Objects	Substitution
Method overriding	Signature Types of results	Classes under Inheritance	Substitution
Strategy pattern	Strategy interface	Algorithms	Substitution
Factory pattern	Abstract Creator and product	Concrete creators and products	Factory
Decorator pattern	Components and decorator interfaces	Concrete components and decorators	Composition
Template pattern	Template of a method	Method steps	Template

As *vp*-s with variants become much more than places where some variability happens, we propose a new definition, which extends Definition 1.

Definition 3 In object-oriented systems, variation points with variants abstract the structure (*a.k.a.*, design) and the functionality of the implemented variability. They represent the unchanged parts and parts that change, that is, local symmetries, in software design that are realized by traditional techniques.

According to Figure 2 and Definition 3, *vp*-s with their variants mark local symmetries in code assets. But, this does not imply that all local symmetries in these techniques denote also variability implementations. Hence, local symmetries in code assets are merely potential candidates to be *vp*-s with variants.

Remark 1 In the following, most often we will use *variation points* (*vp*-s) and *variants* instead of the long name *candidate variation points and candidate variants*. Their difference is important only in Section 8, where we distinguish which of the candidate variation points and variants are relevant.

To illustrate the deduced interrelationship in Figure 2 and its subsequent Definition 3, the existing *vp*-s with variants in Listing 1 can be identified by simply identifying the local symmetries in it. Based on Figure 1, the first identified local symmetry is in inheritance, which implies the `vp_Shape` variation point (lines 1–10) with its variants, `v_Rectangle` (lines 11–32) and `v_Circle` (lines 33–43). Whereas, the second identified local symmetry is in overloading, which implies the `vp_Draw` variation point (lines 21–31) with `v_drawCoordinates` (lines 22–26) and `v_drawPoint` (27–31) variants. In addition to *vp*-s, identifying the variants of a *vp* is important, as they may have nested variability. For example, the class `Rectangle` is a variant of `vp_Shape` but has a nested variation point, `vp_Draw`, which has two other variants.

To automate the identification of *vp*-s with variants, we summarize in Table 1 eight common software constructs and their elements of symmetries, that is, their unchanged and changed properties of software under their specific

symmetry transformation. These data are based on existing studies and the way to interpret symmetry on software constructs [16, 99, 100, 98]. This could be extended to include symmetry in other language constructs and software design patterns. Moreover, all eight software constructs in Table 1 have symmetry at the class or method level, indicating that any identified *vp* or variant implemented by these techniques will have a class or method level granularity. Then, distinguishing the unchanged and changed parts in a software construct, as in Table 1, is indeed decisive for automating the identification process. The first step of the approach thus relies on the identification of local symmetries in these software constructs, which represent candidate *vp*-s with variants.

4.2 Density of candidate vp-s with variants

After the identification process, our approach aims also at facilitating the variability comprehension of the considered system.

According to Alexander's theory, the number of local symmetries is crucial for measuring the coherence of a structure [1, 2]. Namely, their high density makes easier to recognize, describe, and remember a given structure. Similarly, we propose to use the density of candidate *vp*-s with variants to easily locate and describe the most intense places with variability in a system, as a way to analyse and comprehend its variability. This extrapolation is feasible because of the nested nature of candidate *vp*-s, which corresponds to the recursive nature of centers in Alexander's meaning. For example, in Listing 1, the `vp_Draw` is a nested *vp* of the `vp_Shape`, by being within one of its variants. This density indicates the amount of variability that is concentrated in the code assets of Listing 1. Therefore, we give the following definition.

Definition 4 Variability density is the amount of local symmetries, as candidate *vp*-s with variants with their nested and related candidate *vp*-s with variants, within a code unit or within a given part of code assets.

A quantification of this variability density is possible. It represents the number of class level *vp*-s and variants related under the inheritance relationship and the number of method level *vp*-s and variants within those *vp*-s. For example, in Listing 1, there is one class level *vp* and two variants (related under inheritance relationship), which have another method level *vp* with two variants also (as nested within the `v_Rectangle`). In this case, the variability density is 6. Herein, instead of quantifying this density, we presume that within a given part of code assets variability density can be visually perceived. For example, the relative amount of local symmetries for the given part of code assets in Listing 1 should be directly discerned. In this way, the comprehension of a system's variability can be simplified by using a visualization form that will enable us to directly discern places with different variability densities.

5 Automatic identification and visualization of local symmetries

To show the feasibility of our symmetry-based approach, we developed the *symfinder* toolchain. In the following we detail its support for automatic identification of local symmetries in code assets of a system and their visualization.

5.1 The *symfinder* toolchain

Following our approach, *symfinder* employs symmetry in six software constructs to identify *vp*-s with variants in a system and then visualizes them in a way that highlights their density. Figure 3 depicts the whole dockerized toolchain of *symfinder*, which is publicly available[4]. It consists of three main steps, corresponding to the blue boxes:

(1 & 2) It first fetches the sources of the targeted variability-rich system that is shared on a *git* software-hosting platform.

(3 & 4) Then, the *symfinder* engine translates the relevant parts of the code assets into elements in a Neo4j[5] graph database, and queries the graph nodes using the Cypher language[6] to identify local symmetries in each technique based on their defined commonality and variability (*cf.* Table 1).

(5) Finally, it visualizes the identified local symmetries (*i.e.*, candidate *vp*-s with variants) through an appropriate visualization embedded in a web browser.

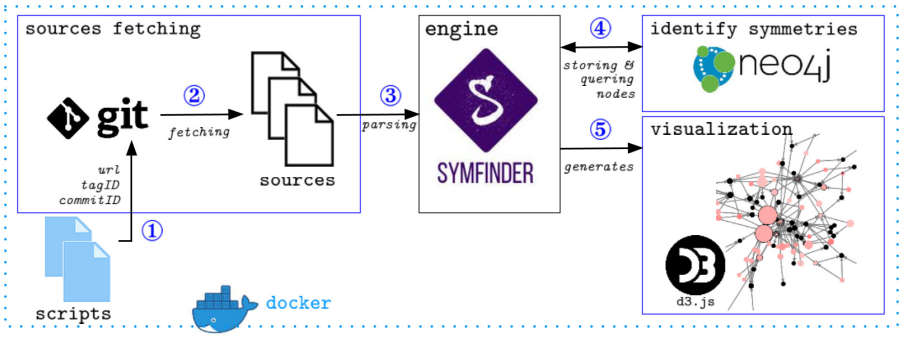


Fig. 3: The dockerized *symfinder* toolchain

[4] https://github.com/DeathStar3/symfinder
[5] https://neo4j.com/
[6] https://neo4j.com/developer/cypher/

The toolchain uses several scripts, an engine implemented in Java, and the Neo4j graph database. To increase its portability and facilitate its usage, it is deployed within a Docker[7] container.

As the identification (3 & 4) and visualization (5) steps constitute the main logic of *symfinder*, they are detailed in the following subsections. On the other hand, the sources fetching step (1 & 2) of the toolchain mainly aims at automating the experiments. From a configuration file, *bash* and *python* scripts are run to fetch sources and checkout the desired tags or commits from some *git* repositories (*cf.* Figure 3). This enables *symfinder* to work easily over any project that is publicly available on a software-hosting platform (*e.g.*, GitHub or GitLab). More details on the internal project structure of *symfinder*, with usage guidelines, are given in a companion page (see Section 6.3).

5.2 Automatic identification

At the center of the toolchain in Figure 3 is *symfinder* engine. Its main purpose is to automatically analyse the source code of a targeted software system, to identify *vp*-s with variants, and to build a visual representation of them.

Local symmetries are identified according to the defined symmetry in two first language construct and four software design pattern given in Table 1, that is, their unchanged parts and parts that change are identified. Specifically, each interface, abstract class, extended class, overloaded constructor, overloaded method, and unchanged part in four design patterns is identified. All together, they actually represent *vp*-s. Then, the classes that implement or extend them, including the concrete overloaded constructors and methods, and the parts as changes in four design patterns are also identified. They represent the respective variants of each *vp*.

Technically, the identification process is made of three steps. First, the source code is parsed and the structure of the implementation units of the analysed system is stored into the Neo4j graph database, where each class, interface, method, and constructor is represented by a node, including the structural relationship of these nodes (*e.g.*, a method belongs to a class). In the version of *symfinder* we used for our experiments, Java was the only supported language[8]. The Eclipse JDT parser is used in it to analyse Java classes. In this step, nodes and their relationship types are queried by the Cypher language and labeled, namely `CLASS`, `ABSTRACT`, and/or `INTERFACE` for nodes, `EXTENDS` or `IMPLEMENTS` for inheritance relationships.

Secondly, we identified local symmetries in four design patterns, listed in Table 1. Considering the state of the art methods on software design pattern detection, such as structural or behavioral analysis methods [83, 35, 21] using ASTs (Abstract Syntax Tree) [71] or graph representations of the codebase [96], we decided to use the graph representation of the structure of the

[7] `https://www.docker.com/`

[8] A version supporting both Java and C++ is also available at `https://deathstar3.github.io/symfinder-demo/splc2020.html`.

implementation units (*i.e.*, classes and interfaces) and to rely on its structural analysis. This is not as precise as a behavioral analysis, but is sufficient to identify local symmetries in basic instances of design patterns on larger systems. Although design patterns mainly rely on inheritance, they also make use of finer-grained elements, for example, a strategy is used as a field in another class, and a factory uses methods return types. Hence, identifying local symmetries in such design patterns implies being able to detect such elements and resolve their full class name to determine if they are part of a design pattern. In this step, the resulting graph in the Neo4j database is queried again using the Cypher language, during another analysis of the codebase, to extract these elements and identify local symmetries in design patterns[9]. Doing so allows us to search for an exact graph match on subgraphs containing a limited number of nodes, thus reducing the complexity [92] [10]. In this step, nodes that correspond to the unchanged part of the four design patterns are labeled with their respective name, that is, `STRATEGY`, `FACTORY`, `DECORATOR`, or `TEMPLATE`.

In the last step of the identification process, the existing labeled nodes and their relationships in the graph database are queried using Cypher so that local symmetries, as candidate *vp*-s with variants, are identified. During this step, nodes representing interfaces or abstract classes, or classes being extended, or being the unchanged part of a design pattern, are labeled as `VP`s. Then, classes or interfaces implementing or extending these *vp*-s are labeled as `VARIANT`s.

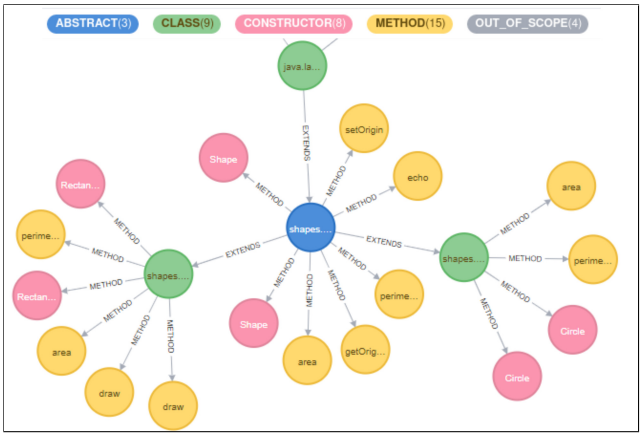


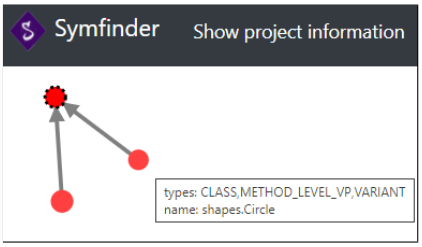
Fig. 4: The translated code of Listing 1 to a Neo4j graph

[9] This second parsing stage is needed as, due to limitations of Eclipse JDT, the structure of the whole codebase is needed in the database in order to query it and determine the correct types of each element.

[10] Our identification of local symmetries in software constructs is using a graph representation of the codebase, but it must not be confused with the graph symmetry detection or automorphisms [61].

```
/* Identifying variants of a variation
point realized by subtyping */

MATCH (sc:VP)-[:EXTENDS|IMPLEMENTS]->(c)
WHERE c:CLASS OR c:INTERFACE
SET c:VARIANT
```



a: Cypher query to identify variants in suptyping b: Generated visualization for Listing 1

Fig. 5: The used query to identify class variants and their visualization

For example, Figure 4 shows the Neo4j graph obtained by analysing the source code in Listing 1. The nodes `shapes.Rectangle`[11] and `shapes.Circle` are labeled with `CLASS`, the node `shapes.Shape` as `ABSTRACT`, and their relationship as `EXTENDS`. Then, using the Cypher query as in Figure 5a, the nodes `shapes.Rectangle` and `shapes.Circle` are identified and so labeled as `VARIANT`s of the node `shapes.Shape`, which is already identified by another query as a `VP`. All used Cypher queries are based on Table 1, which we have also documented and are available online [12]. It should be noted that Neo4j is used only as a database and not as a visualization tool (as detailed in Section 5.3.2).

5.3 Visualization

In the step 5 in Figure 3, *symfinder* generates a visualization with only the identified local symmetries stored in the database.

5.3.1 Visualization principles

Two following demands guide the organization of the provided visualization.

Demand 1 *One should easily discern zones of interest with regard to variability in the observed system.*

To meet this demand, we exploit the analysed local symmetries and the general notion of the density of *vp*-s with variants (*cf.* Section 4.2) to visualize the variability of a system. As local symmetries are mainly organized around inheritance, we decide to architecture the visualization by focusing on classes, being nodes, in their inheritance graph, with edges representing inheritance links (*i.e.*, `extends` and `implements` in Java).

Going beyond plain nodes, we also choose to visualize information regarding the used language constructs and design patterns for implementing

[11] The `shapes` in this path is the package's name in Java.

[12] https://deathstar3.github.io/symfinder-demo/identification-method.html

Table 2: Ten kinds of nodes and their relationships used for the visualization

Node types	Node meaning	Parameters	Visualization
Concrete class	Variation point	Node with black outline	
Concrete class	Variant with inner *vp*-s [1]	Node without an outline	
Abstract class	Variation point	Node with dotted outline	
Interface	Variation point	Black node	
Constructors	Variation point	Node with shades of red	
Overloading	Variation point	Node of different size	
Strategy pattern	Variation point	Node with symbol **S**	
Factory pattern	Variation point	Node with symbol **F**	
Decorator pattern	Variation point	Node with symbol **D**	
Template pattern	Variation point	Node with symbol **T**	
Inheritance	Edge		

[1] It can be neither a *vp* nor a variant but simply a class with inner *vp*-s at method level, or a class variant without method level *vp*-s, if all variants are displayed

each *vp* with its variants. As in many software and code artifacts visualizations [48, 49, 95, 94, 90], we rely on the visual principles of preattentive perception [22], 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. In Table 2 are shown the ten different kinds of nodes that are used in *symfinder* to visualize the used techniques for realizing *vp*-s with variants.

As a result, a class level *vp* can be more distinguishable by other method level *vp*-s (*e.g.*, through its size and intensity of colors) or if it is connected by inheritance to other *vp*-s or variants. They will also all together form a more noticeable zone in the graph, showing the density of *vp*-s and variants. It should be noted that the visualization approach is based on the concept of density and not on symmetry. Symmetry in software constructs is identified, their inheritance relationship and density are visualized, but, one should not confuse the two and expect any kind of symmetry in the visualization.

Demand 2 *One should be able to gain a general view of the amount of variability in the observed system and a specific view of each identified vp with its variants.*

As discussed in the previous paragraphs, the graph-based representation already provides a general view of the variability based on its density. To improve it, we also display the total number of *vp*-s and variants, at class and method levels. Then, we simply apply the classic *visual information seeking mantra* of [84]: overview first, zoom and filter, then details-on-demand. We first improve the visualization of nodes to denote the used design patterns by their first letters. As we consider the overview complete, we add the zoom in/out option so that a specific area of visualization with potential *vp*-s and variants can be magnified. We also provide a way for filtering out solitary nodes, as the obvious least dense part of the graph, or any given node by its

name, so in the visualization remains only those *vp*-s and variants that are most likely to interest the user. Finally, while hovering over a node, the name of each *vp* and variant at class level is visualized, as well as the label of the node (*e.g.*, `VP`, `METHOD_LEVEL_VP`, or `STRATEGY`).

5.3.2 Implementation

Although we considered using the visualization capabilities of Neo4j and other visualization forms used in SPL engineering [54], we decided to use the D3.js library[13]. It indeed allows for the visualization of highly customizable forms of graphs, so to meet both demands mentioned above, but also a plethora of chart types and visualization forms that help us in experimenting before devising the current graph flavor and could help in future evolution of the toolchain.

Besides, as D3.js visualizations are written in JavaScript, only a web browser is needed for display, facilitating portability. The implementation simply uses JavaScript configuration files in a template for the web page that will display the density of *vp*-s with variants as disconnected graphs.

5.3.3 Illustration

For completeness, Figure 5b shows the generated visualization for the example in Listing 1. Although, to illustrate most of the provided options in the visualization, Figure 6 shows an excerpt of the visualization with the identified local symmetries by *symfinder* in JFreeChart 1.5.0, a variability-rich system among the ones we used as subjects in our experiments described in Section 6.

As a fulfillment of Demand 1, its visualization shows that several zones with different densities of *vp*-s with variants can be easily discerned in JFreeChart. Following our approach (*cf.* Section 4.2), the comprehension of variability for a system can start from places with a higher density of variabilities, as potential zones of interest. Such is the part of the graph surrounded with a blue rectangle in Figure 6, which is manually added to the screenshot. The corresponding magnified view is given in Figure 7.

Regarding Demand 2, the `'Show project information'` menu provides overall numbers. In JFreeChart 1.5.0, 924 *vp*-s with 1,925 variants are identified (*cf.* as given latter in Table 3). Then, each *vp* with its variants is visualized by a circle that points out the used implementation technique (*cf.* Table 2). Specifically, a red node without an outline is a concrete class that represents a variant with variability at the method level, such as the `v_PolarPlot` in Figure 7. A red node with a dotted outline visualizes an abstract class, whereas a black node an interface. Both of them represent a *vp*, such as the `vp_Zoomable`. Multiple shades of red nodes are used to visualize the number of constructor overloads for each class or interface, that is, method level variability. The more overloaded constructors are present, the more intense is the node's color. Next, the size of the node is in the function of the number of overloaded methods.

[13] `https://d3js.org/`

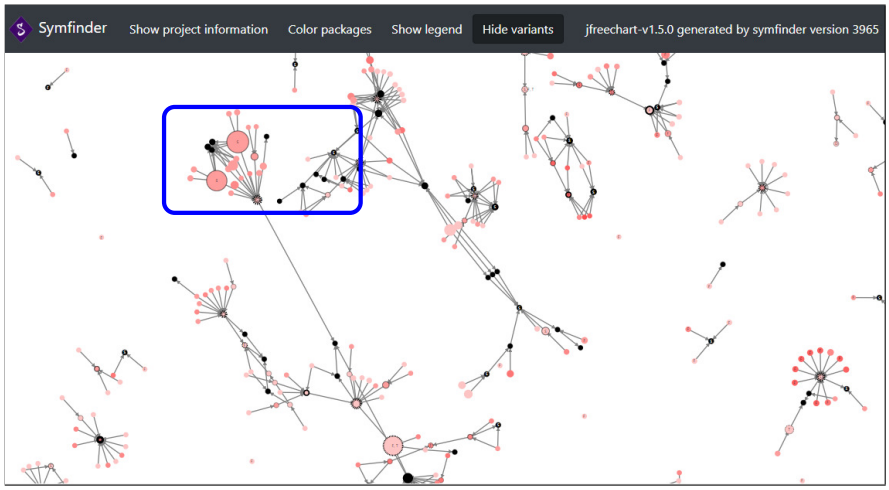


Fig. 6: An excerpt of the JFreeChart 1.5.0 visualization

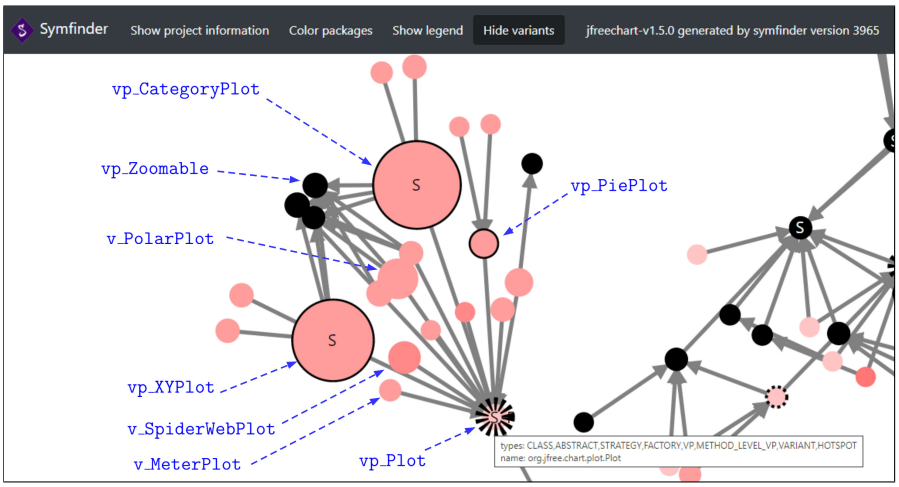


Fig. 7: *vp*-s with variants in JFreeChart 1.5.0 for the selected zone in Figure 6

For instance, the node `vp_XYPlot` has a larger size, indicating that it has variability at method level. Further, the first letter of a design pattern is used to mark a *vp* implemented by that pattern, for example, letter `S` is used for the strategy pattern in `vp_Plot` and its dotted outline denotes its relation to an abstract class. All these information are available from the `'Show legend'` menu in visualization.

In addition, the zoom functionality is illustrated in Figures 6 and 7. As for filtering, the `'Show/Hide variants'` menu makes possible to show or hide the class variants without method level variability, as places with the lowest den-

sity of variability in the visualization. The `'Show project information'` also provides an option to filter out *vp*-s with variants within any specific package or class the user is not interested in. Finally, the label of a *vp* or variant to its respective class in code assets appears when hovering the node, such as the label for the `vp_Plot` to its `types` and `name: org.jfree.chart.plot.Plot` or for the variant of `Circle` in Figure 5b. It must be noted that blue names and arrows in Figure 7 have been manually added to the screenshot, while only the label for the `vp_Plot` is shown on this illustration. The whole visualization for the JFreeChart 1.5.0 is also available in the companion page.

6 Evaluation design

We now describe the designed evaluation of the proposed approach, defining the goal and research questions, and the selected subject systems.

6.1 Goal and research questions

Following the Goal-Question-Metric process [10], we setup the following goal: automate the identification of *vp*-s with variants in a real variability-rich system, in a scalable way, and provide a handy representation of them to users that want to comprehend the system's variability. To address this goal, we define three research questions.

RQ_1 : **What is the amount of identified variability in a real variability-rich system?** We first investigated the amount of identified variability by *symfinder* in real variability-rich systems. Then, we investigated the tool's capabilities to run in different environments and its scalability during the identification and visualization phases. To this end, we used sixteen subjects, all of them being real open-source systems implemented in Java. For each of them, we counted the total number of *vp*-s with variants and measured the *symfinder*'s execution time.

RQ_2 : **To what extent the identified local symmetries in a system are *actual* variation points with variants, that is, are relevant to its domain features?** During the presentation of our approach in Section 4.1, we explicated that *vp*-s with variants mark local symmetries in code assets, but whether its reverse is true it needs to be proved. Therefore, we investigated whether the identified local symmetries in a system, often referred here as candidate *vp*-s with variants, are indeed variability related, that is, whether they are relevant to the system's domain features. Additionally, in case that there are irrelevant local symmetries, we also analysed how to distinguish them in the visualization. In such a way, we aim to measure the *precision* and *robustness* of our approach.

RQ_3 : **To what extent *symfinder* can be used by software architects to comprehend the variability of their own system?** With only a prototyped implementation, it is hard to gather user's feedback, but we

still aimed to evaluate the approach from a user's perspective. We thus investigated whether *symfinder* can be helpful to a real software architect to understand the implemented variability in his own system. We report here a subjective evaluation from the experience of Daniel Le Berre, co-author of this paper and also software architect of Sat4j[14], when he used *symfinder* on his system.

6.2 Subject systems

To addresses the three given research questions, we applied *symfinder* to sixteen Java-based variability-rich systems, as evaluation subjects. In the following, we present these systems, while the experiments and obtained results for each research question are presented in Sections 7, 8, and 9, respectively.

6.2.1 Selection criteria

For selecting the sixteen evaluation subject systems, we considered several criteria: their proximity to a real-world software system, their implementation in Java, the open-source nature of the project, their availability on a *git* repository, and the fact that they could contain some implemented variabilities. Then, some of them should have a ground truth with the traces of domain features to code assets, so to tackle RQ_3. Due to the rarity of such data sets, we will report on an experiment conducted on two systems in that particular case. In a complementary way, at least one of the considered systems should be studied by someone that has firsthand knowledge of its variability domain and implementation, ideally by one of its main software architects, so to tackle RQ_3. Here, we will report on a single system with such knowledge.

By the first criterion, we aim to evaluate our tooled approach in a real-life context, thus providing an ecological validity [41]. Whereas, by the rest of the criteria we aim to make possible the replication of our evaluation or extend it with new subject systems.

6.2.2 Description of selected subject systems

We first selected seven systems that are used in some previous research works in SPL engineering (*e.g.*, JHipster by Halin et al. [33]) or in our first publication on *symfinder* (such as JavaGeom [89]). They are: Java AWT, Apache CXF, JUnit, Apache Maven, JHipster, JFreeChart, and JavaGeom. Then, we added seven new systems. For the time frame of 10 last years, they are under the most starred Java projects on GitHub, with a number between 1,797 and 48,838 stars. Namely, Deeplearning4j, Elasticsearch, Jackson-Core, ZXing, Mockito, RxJava, and Guava. A brief description of them is also given in Table 3.

[14] http://www.cril.univ-artois.fr/~leberre/

Finally, we selected two systems to handle the experiment with domain features. The first one is ArgoUML, which is used in different studies on SPL engineering [18, 19, 67, 70]. It has a ground truth with traces of domain features to code assets [59]. Then, we selected Sat4j, a Java library for solving boolean satisfaction and optimization problems such as SAT, MAXSAT, Pseudo-Boolean, and Minimally Unsatisfiable Subset (MUS) problems. It is used since 2008 in the Eclipse platform to manage its plugin dependencies [52]. Sat4j has already been used as a benchmark for automatic testing approaches [39, 93]. Then, its software architect is one of the authors in this paper, who has the knowledge for its architecture, its variability domain, and its realization in the code.

6.3 Availability of systems and experiments

Details of these sixteen subject systems are presented in the first three columns in Table 3, namely, the URL to their public repository, the analysed tag or commit ID, and lines of code (LoC). All conducted experiments included in this paper are also available at https://deathstar3.github.io/symfinder-demo/jrn20.html, with extracted screenshots, more explanations on each case, and a deployed online demonstration of the visualization.

7 Amount of variability and *symfinder*'s scalability in real systems

In this section are presented the conducted experiment, our observations, and the gained results concerning the first research question, RQ_1.

7.1 Conducted experiment

By this experiment we want to reveal the amount of variability in real variability-rich systems and whether our toolchain was able to successfully identify it within a reasonable time.

We thus applied the *symfinder* toolchain in each of the sixteen subject systems. Table 3 gives the analysed LoC [15] for each system. It can be noted that the selected set of subject systems help us to reveal amounts of identified variability in real variability-rich systems of different sizes. To evaluate the interoperability of our tool, we run the same experiments on three operating systems, Linux, Mac, and Windows. Finally, we recorded the execution time taken to get through the whole toolchain up to the generation of the visualization, making a first evaluation of the scalability of *symfinder*.

Table 3: Sixteen variability-rich systems with their respective LoC, total number of candidate *vp*-s with variants, and their class or method level granularity

Subject system			Class (C) and Method (M) level			
Description	Tag/Commit	LoC		#vp-s	#variants	#nodes
Java AWT	jb8u202-b1468	69,974	C	223	175	33
API to develop GUI			M	572	1,531	–
Apache CXF 3.2.7	cxf-3.2.7	48,655	C	1,134	1,447	509
web services framework			M	2,265	6,173	–
JUnit 4.12	r4.12	7,717	C	44	60	14
unit testing framework			M	65	185	–
Apache Maven 3.6.0	maven-3.6.0	105,342	C	268	242	104
build automation tool			M	341	906	–
JHipster 2.0.28	2.0.28	2,535	C	39	12	2
application generator			M	13	44	–
JFreeChart 1.5.0	v1.5.0	94,384	C	257	277	45
charting library			M	667	1,648	–
JavaGeom	7e5ee60	32,755	C	74	52	14
library of geometric shapes			M	262	707	–
ArgoUML	bcae373	134,367	C	327	860	85
UML diagramming application			M	447	1,116	–
Deeplearning4j 1.0.0	deeplearning4j-1.0.0-beta5	1,030,214	C	322	659	317
library for deep learning			M	1,317	3,421	–
Elasticsearch 6.8.5	v6.8.5	683,527	C	1,008	1,782	398
search engine			M	3,090	7,253	–
Jackson-Core 2.10.1	jackson-core-2.10.1	3,042	C	43	25	24
streaming API			M	250	732	–
ZXing 3.4.0	zxing-3.4.0	33,103	C	36	98	30
barcodes scanning library			M	89	196	–
Mockito 3.1.12	v3.1.12	16,384	C	142	176	28
mocking framework for unit tests			M	96	308	–
RxJava 2.2.15	v2.2.15	89,221	C	184	854	35
library for composing programs			M	415	1,379	–
Guava 28.1	v28.1	87,802	C	358	238	131
Google core libraries for Java			M	720	1,886	–
Sat4j	22374e5e	27,638	C	80	135	10
library for Boolean problems			M	188	453	–

7.2 Amount of variability

In order to give an insight regarding the amount of variability in a real system, we decided to observe its identified number of *vp*-s with variants, while distinguishing their class and method level granularity. The calculation of these two metrics is automated within the *symfinder* toolchain and they are available from the `"Show project information"` menu in visualization.

7.2.1 Number of vp-s with variants

Interestingly, a recent literature review on metrics in SPL engineering shows that the number of *vp*-s is a useful metric for analyzing variability and its implementation in code [25]. It is used to measure the total number of `#ifdef` – blocks when preprocessors are used to implement variability. Similarly, we

[15] For counting the lines of code we used gocloc: `https://github.com/hhatto/gocloc/`

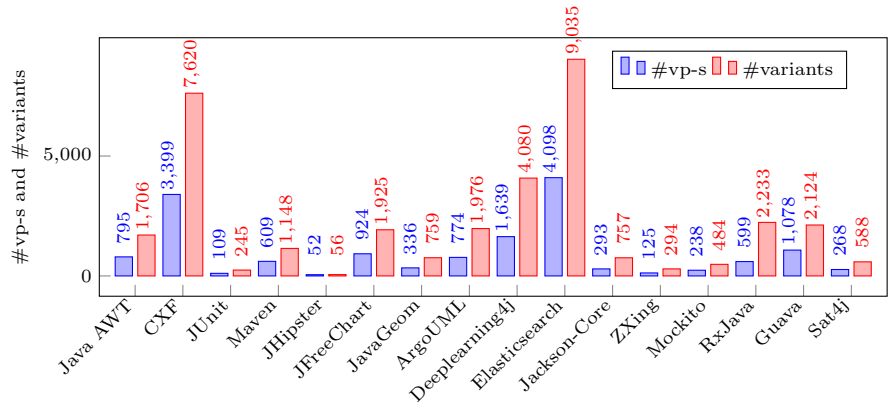


Fig. 8: The total candidate #vp-s and #variants in each subject system

use this metric to reason on the amount of implemented variability in our targeted systems. In contrast to the existing usage in other approaches, but in accordance with our *vp* definition (*cf.* Definition 3), the number of *vp*-s now represents the number of local symmetries in code assets and is complemented with the number of their variants.

The three last columns in Table 3 and Figure 8 show the resulting number of *vp*-s and variants that are identified by *symfinder* in the sixteen subject systems. For instance, the smallest analysed system, JHipster with 2,535 LoC, has 52 *vp*-s that in total have 56 variants. Whereas, the largest one, Deeplearning4j with over 1 million LoC, has 1,639 *vp*-s, which have in total 4,080 variants. Still, it has almost half less *vp*-s than Elasticsearch, which is of far smaller size. When looking at these systems, we observed that Elasticsearch is actually handling much more variability in its implementation than Deeplearning4j.

As a metric, the number of *vp*-s with variants seems useful on this sample of systems to have an overall perception of the amount of their variability.

7.2.2 Granularity of vp-s with variants

We recorded the granularity of *vp*-s with variants, as we consider it important for variability management, especially to trace variability in order to maintain and resolve it.

The last columns in Table 3 show the number of *vp*-s with variants at class and method levels that are identified in each system. The `#nodes` column represents the number of classes that are not *vp*-s or variants but have method level *vp*-s, therefore are identified and visualized. In all systems, some class level variants are the common part, that is, a *vp*, for some other variants. They are known as nested *vp*-s [88], for example, in Jackson-Core it looks like there are more class level *vp*-s than variants, but 19 variants are also nested

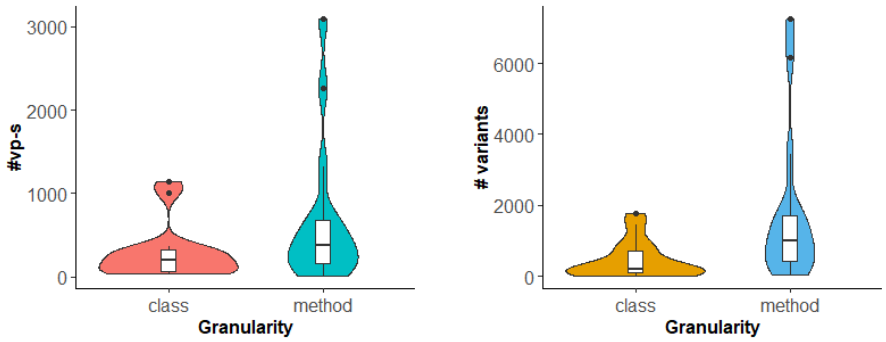


Fig. 9: The distribution of *vp*-s and variants at class and method levels

vp-s. Thus, following the logic of compound features in a feature model and for simplification, we categorized them as *vp*-s.

Based on the values in Table 3, in Figure 9 is plotted the distribution of *vp*-s and variants at class and method levels for all the systems. The distinction between class and method levels enables to deduce some interesting findings. In general we can observe that there are always more *vp*-s and variants at the method level than at the class level. Still, particular systems can be an exception to this, regarding the number of *vp*-s. For instance, JHipster and Mockito have more *vp*-s at the class level than at the method level. This holds even if we remove from consideration the nested *vp*-s, that are added to the class level $\#vp - s$ values in Table 3. But, a possible reason for this could be that JHipster is a server-side library used by the JHipster Generator which is written in JavaScript, while we have analysed only its Java implementation part. Then, few method level *vp*-s are expected for Mockito as its implemented features are barely variable.

Besides, we also observed how this granularity is reflected in the visualization by *symfinder* for each system. Interestingly, among all subject systems, the visualization in Maven gives the impression that it has more class level than method level *vp*-s, which contradicts the shown data in Table 3. This is due to classes with method level variability, which are denoted as `#nodes` in Table 3. However, from Table 3 and Figure 9, in all subject systems, there are always far more variants at the method level than at the class level.

This study of the granularity indicates that both techniques at class and method levels seem to be applied to implement variability, while those at method level are more extensively used. This also confirms the versatility in the implementation techniques and the complexity we tackle in this work.

7.3 Scalability of *symfinder*

The *symfinder* toolchain, presented in Section 5, is designed to identify and visualize variability implementations in large code bases, where a manual anal-

Table 4: Properties of our test system

System	Properties
Processor (CPU)	Intel Xeon CPU E5-2637v2, 3500 MHz, L1-Cache 16KiB
RAM	512GB RDIMM LV 1600MHz
Operating System	Ubuntu 18.04.2 LTS (Bionic Beaver) 64-Bit
Filesystem	EXT4
Hard Disk	250 GiB RAID5
Java Runtime Environment (JRE)	OpenJDK Runtime Environment 1.8.0_201-b08
Java Virtual Machine (JVM)	OpenJDK 64-Bit Server VM 1.8.0_201

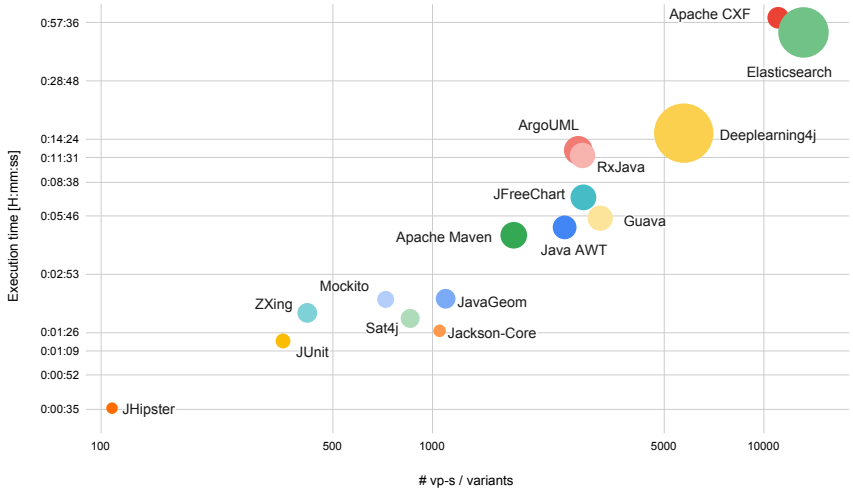


Fig. 10: The *symfinder*'s execution time in terms of *vp*-s. The size of the bubbles represents the number of lines of code (LoC) of the subject system

ysis is not viable. To fulfill this goal, the toolchain has to be able to analyze code bases of thousands of lines of code (LoC) and provide a visualization with all identified *vp*-s and their variants, which also can be thousands of them.

7.3.1 Experiment setup and protocol

In order to make a first assessment of the scalability of our toolchain, we run *symfinder* on our sixteen subject systems, which have different sizes (*cf.* LoC in Table 3), and measured the evolution of the execution time depending on the number of lines of code and the number of identified *vp*-s with variants in each system. Our test environment is a virtual machine deployed on a local network for the sole purpose of *symfinder*. Table 4 details the setup of the system we used to run the experiments. Analyses are run on each system sequentially, to prevent side effects.

7.3.2 Results on execution time and visualization

Figure 10 summarizes the results of our experiments. The x-axis represents the number of *vp*-s with variants identified by *symfinder*. The y-axis represents the execution time, which corresponds to the time spent during the identification of *vp*-s with variants and does not include the time needed to clone each project. And, the size of the bubble represents the number of LoC in the system.

We can make three observations. First, the time taken to analyse real systems is quite acceptable. Our sixteen analysed systems took between a half minute, for JHipster, and an hour-long, for CXF. Then, the execution time increases quite linearly with the number of identified *vp*-s with variants. For instance, ZXing with 419 *vp*-s /variants took few less seconds to be analysed than Mockito with 722 *vp*-s /variants. But, the same cannot be said for the rate between the execution time and the analysed LoC. For instance, by comparing the execution time for CXF, Elasticsearch, and Deeplearning4j, with their analysed LoC (bubble size), we can observe that the execution time has a considerable disproportion with the LoC. These three observations show that *symfinder* is able to identify *vp*-s with variants in projects with above one million lines of code within a reasonable time. The execution time seems to depend in particular upon the number of identified *vp*-s with variants and less upon the analysed number of LoC.

Our visualization is also able to scale on such projects. For instance, the generated visualization of Elasticsearch[16] displays 2,790 nodes representing all its identified *vp*-s with variants at the class level. In the center of the visualization, one can discern zones of a high density of variability. Then, as the generated visualizations are embedded in HTML pages, they are easily deployable online and need only a web browser to be viewed. Moreover, we have successfully displayed them using Mozilla Firefox, Google Chrome, and Safari on Windows, MacOS, and Linux.

7.4 Summarized answer to RQ1

As an answer to the first research question (RQ_1), the results of our different experiments show that real variability-rich systems (with 2K - 1M LoC) have a considerable amount of variability (between 108 and 13K *vp*-s and variants), indicating that its manual identification, or their trace and refactoring to domain features, are expensive activities that will require cumbersome effort. Hence, an automatic identification and visualization approach, such as *symfinder*, can ease the fast identification (within a reasonable time, between 35 seconds to 1 hour) of potential variability places in the targeted system.

[16] `https://deathstar3.github.io/symfinder-demo/JRN20/standard_version/elasticsearch-v6.8.5.html`

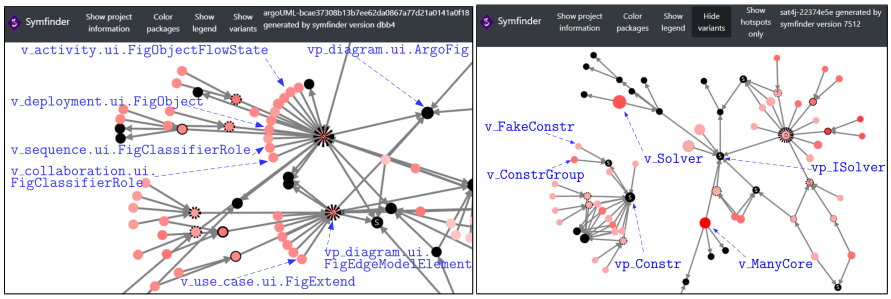


Fig. 11: An excerpt of the visualization in ArgoUML (left) and Sat4j (right)

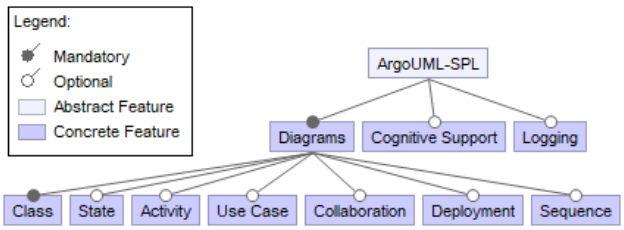


Fig. 12: Feature model of ArgoUML, adapted from [18]

8 Evaluation of the actual vp-s with variants

To address RQ_2, we have to demonstrate that the identified local symmetries in code assets of the targeted systems, referred to as candidate *vp*-s with variants, are actually *vp*-s with variants. To do so we conducted two experiments in ArgoUML and Sat4j, which are detailed in Sections 8.1 and 8.3, respectively.

8.1 Experiment for measuring precision and robustness of our approach

As we aim to gain insights into the precision and robustness of our approach, we first prepared the needed ArgoUML's and Sat4j's artifacts. Then, we evaluated how many of their identified local symmetries by *symfinder* have a mapping to their domain variability. To do this, we defined and automatically measured precision and recall over this mapping.

8.1.1 Artifacts preparation

ArgoUML's artifacts. As a first subject system for this experiment we used ArgoUML. To the extent of our knowledge, it is one of rare publicly available Java-based variability-rich systems that provides a feature model (FM) [18] and a ground truth with the traces of its optional features to the code assets, which are annotated [59]. Based on a recent and continuously updated catalog

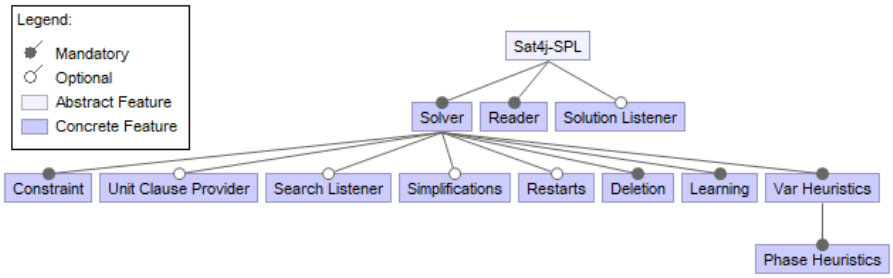


Fig. 13: Feature model of Sat4j

with 131 case studies used in extractive SPL approaches[17] [57], ArgoUML is the only one that is a real Java-based system with an available ground truth.

Extracted by Couto et al. [18], ArgoUML's FM consists of 11 features, given in Figure 12. The abstract feature `ArgoUML-SPL` represents conceptually the ArgoUML's variability domain. It has 2 mandatory features, `Diagrams` and `Class`, and 8 optional features, `State`, `Activity`, `Use Case`, `Collaboration`, `Deployment`, `Sequence`, `Cognitive Support`, and `Logging`. In the ArgoUML's ground truth [59], each of these 8 optional features has a set of traces to code assets. In total there are 714 features traces (without duplication).

Our application of *symfinder* to ArgoUML identified 1,272 candidate *vp*-s with variants at class level, some of which have candidate *vp*-s with variants at method level (*cf.* Table 3). A visualization excerpt of the identified variability in ArgoUML by *symfinder* is given in Figure 11 (left)[18].

Sat4j's artifacts. We then used Sat4j as a second subject for this experiment. Its software architect prepared its ground truth for the purpose of this study. He manually set up the Sat4j's domain features into a feature model, given in Figure 13, and their traces to code assets. To accomplish this, using Java annotations, he annotated those classes, interfaces, methods, or fields in the `org.sat4j.core` that belong to each domain feature[19] Then, all annotations are extracted using an internal utility tool. As an output, it provides the list of traces into a markdown file[20].

Sat4j's feature model consists of 13 features. The abstract feature `Sat4j-SPL` represents conceptually the Sat4j's variability domain, which has 7 mandatory features, `Reader`, `Solver`, `Constraint`, `Deletion`, `Learning`, `Var Heuristics`, and `Phase Heuristics`, and 5 optional features, `Solution Listener`, `Unit Clause Provider`, `Search Listener`, `Simplifications`, and `Restarts`. In

[17] The catalog: `https://but4reuse.github.io/espla_catalog/`. According to its our last visit on November 20, 2020.

[18] The whole ArgoUML's visualization is available at `https://deathstar3.github.io/symfinder-demo/JRN20/hotspots_version/argoUML-bcae37.html`.

[19] Sat4j's code: `https://gitlab.ow2.org/sat4j/sat4j/-/tree/master/org.sat4j.core`

[20] Sat4j's ground truth: `https://deathstar3.github.io/symfinder-demo/JRN20-files/Features.pdf`.

Sat4j's ground truth, each of these 12 concrete features has a set of traces to its code assets. In total, there are 118 features traces.

Besides, our application of *symfinder* to Sat4j identifies 866 candidate *vp*-s and variants (including `#nodes`), from which 225 are at class level and 641 at method level (*cf.* Table 3). A Sat4j's visualization excerpt with the candidate *vp*-s with variants identified by *symfinder* is given in Figure 11 (right)[21].

8.1.2 Availability of artifacts and the automated mapping

After preparing the ground truths for both ArgoUML and Sat4j, we normalized their data and automated the mapping[22] of candidate *vp*-s with variants to domain features using traces in their respective ground truth. For instance, looking at the names of candidate *vp*-s and variants in Figure 11, they are all expected to have a mapping to domain features in Figures 12 and 13, respectively. Namely, judging by their names, variant `v_sequence.ui.FigClassifierRole` in ArgoUML is expected to be mapped to its `Sequence` feature, or the `vp_ISolver` in Sat4j to its `Solver` feature. The complete raw, normalized, and analysed data are available online[23].

While the ArgoUML's ground truth was taken from an external source, the Sat4j's ground truth was prepared internally. To avoid any possible bias and data manipulation, we first held a meeting with Daniel Le Berre, Sat4j's software architect, where we discussed the purpose of the study and the needed data. Then, only the Sat4j's software architect has prepared the ground truth. Another author automated the mappings, including the mapping in ArgoUML, which were crosschecked by the other authors. Then, we adapted two well-known measures, namely *precision* and *recall*, as explained in the next section. Lastly, results were discussed and reported together.

8.1.3 Measures of precision *and* recall

To evaluate the number of local symmetries that are actual *vp*-s with variants, we define *precision* and *recall* measures in our specific context.

Precision. Let be T_{gt} the set of traces for all features given in the ground truth and I_{vp-v} the set of all identified local symmetries by *symfinder*, that is, the set of all candidate *vp*-s and variants. We use *precision* to measure the percentage of identified local symmetries that are actually *vp*-s and variants. Thus, the identified local symmetries that are mapped to the features are true positives (TP), referred to as actual *vp*-s and variants. Whereas, the identified

[21] The whole Sat4j's visualization is available at `https://deathstar3.github.io/symfinder-demo/JRN20/hotspots_version/sat4j-22374e5e.html`.

[22] While the term 'tracing' / 'trace links' is used in the ground truth, we will distinguish from this term in this experiment by using 'mapping' / 'mapping links' for *vp*-s and variants mapped to features, although both of them have the same meaning.

[23] `https://deathstar3.github.io/symfinder-demo/mapping_process.html`

Table 5: Summarized data from the two ground truths and their results

	ArgoUML	**Sat4j**
#domain features	11	13
#features traces (normalized)	672	113
#local symmetries (class level)	1 272	225
True Positives (TP)	561	113
False Positives (FP)	711	112
False Negatives (FN)	111	0
Precision	44.10%	50.22%
Recall	83.48%	100.00%

local symmetries without a mapping are false positives (FP), referred to as irrelevant local symmetries or not related to domain variability. Hence,

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

Recall. We use *recall* to measure the percentage of domain features traces in the ground truth that is used for the mapping of identified local symmetries to features. Thus, traces that are used for the mapping are true positives (TP), whereas those that are not used are false negatives (FN). Hence,

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

8.2 Results of precision and recall

In ArgoUML, from the 1,272 class level candidate *vp*-s and variants that are identified, our mapping tool found that 561 of them have a mapping to at least one feature, whereas the rest 711 of them are without a mapping. Then, from all 672 features traces, 111 of them are not used for the mapping of candidate *vp*-s or variants. As expected from a non-trivial mapping, several candidate *vp*-s and variants of ArgoUML are without a mapping to features in the ground truth, and conversely. Besides, in Sat4j, out of the 225 identified local symmetries at class level in Sat4j (*cf.* Table 3), our mapping solution found that 113 of them have a mapping to at least one feature, whereas 112 of them are without a mapping. Using the defined measurements in Section 8.1.3, we also calculated the *precision* and *recall* for ArgoUML and Sat4j. The obtained results are given in Table 5

The obtained precision from ArgoUML and Sat4j shows that about 44% and 50%, respectively, of identified local symmetries are relevant to the domain features, that is, are actual *vp*-s and variants. The rest, 56% and 50% of them respectively, are irrelevant to the domain features. The obtained recall values shows that about 83% from ArgoUML or 100% from Sat4j of features traces in the ground truth are used for the mapping of local symmetries to features.

As a matter of fact, this considerable number of false positive local symmetries may have an impact on the time spent to distinguish the actual *vp*-s with variants from the irrelevant local symmetries in the visualization. Although expected, we observed three main reasons for such low precision.

Reason 1. The eight optional features used to extract an ArgoUML SPL are coarse grain and are selected by authors based on the ArgoUML domain knowledge [18]. Their study misses some information regarding how complete is the given list of features. This means that the ArgoUML's ground truth may be incomplete, which explains the local symmetries without a mapping. The precision from Sat4j is a bit higher because its software architect was asked to provide as much as possible a complete list of domain features and their trace links to Sat4j's code assets.

Reason 2. The main attention of feature identification and location approaches is the mapping of *variable* features to code assets, hence *mandatory* features are largely sidestepped [45]. This is due to the fact that variable features are the configuration units during the product derivation in an SPL. For instance, in the ArgoUML's ground truth, the `Class` mandatory feature (*cf.* Figure 12) does not have traces to code assets. This is justifiable as ArgoUML is used as a benchmark to evaluate extractive SPL approaches. However, in our symmetry-based approach, some of the identified local symmetries may have a mapping to mandatory features. This further explains the low precision of our tooled approach in ArgoUML and the higher precision in Sat4j. Specifically, unlike ArgoUML, in Sat4j there are 7 mandatory features from the 13 overall features and each of them has trace links to code assets. More precisely, looking at its ground truth, 69% of trace links are for the 7 mandatory features.

Reason 3. It is likely that not all of our identified places with symmetry in code are variability related, as it is the case with the usage of preprocessor directives in C/C++. Specifically, Zhang et al. [97] state that *"from our experience most #ifdef blocks (*e.g.*, 87.6% in the Danfoss SPL) are actually not variability related, but for other purposes such as include guards or macro substitution"*. While the mechanisms are different, in object-oriented variability-rich systems, in addition to implementing variability, the inheritance technique is likely to be mostly used for other reasons, such as for fundamental structuring of domain objects (*e.g.*, in 56% of cases in the ArgoUML and 50% of cases in Sat4j). Furthermore, after we showed the gained precision and recall to the Sat4j's architect, he pointed out that he had decided to annotate only the concrete classes, but not their abstract classes, to avoid making redundant annotations. Then, only the main variability sources have been annotated. For instance, the code includes many examples of one interface with two concrete classes (implementing the null object design pattern in most cases), which are not annotated. After studying the remaining candidate *vp*-s that are counted as false positives, it appears that they are mainly still variability related, but only at the level of internal implementation, and not at the domain level (*cf.*

Table 6: Precision (P) and recall (R) considering different *density threshold* (*i.e.*, regarding the #variants of a *vp*) values in ArgoUML and Sat4j.

Thre-	ArgoUML					Sat4j				
shold	TP	FP	FN	P(%)	R(%)	TP	FP	FN	P(%)	R(%)
Non	561	711	111	44.10	83.48	113	112	0	50.22	100.00
> 2	526	638	146	45.19	78.27	109	91	4	54.50	96.46
> 3	488	554	184	46.83	72.62	97	71	16	57.74	85.84
> 4	440	482	232	47.72	65.48	84	53	29	61.31	74.34
> 5	386	413	286	48.31	57.44	67	35	46	65.69	59.29
> 6	364	379	308	48.99	54.17	56	33	57	62.92	49.56
> 7	311	327	361	48.75	46.28	48	25	65	65.75	42.48
> 8	279	309	393	47.45	41.52	39	15	74	72.22	34.51
> 9	270	297	402	47.62	40.18	38	14	75	73.08	33.63
> 10	264	283	408	48.26	39.29	38	13	75	74.51	33.63
> 15	194	203	478	48.87	28.87	18	5	95	78.26	15.93
> 20	144	179	528	44.58	21.43	2	1	111	66.67	01.77
> 30	138	132	534	51.11	20.54	No local symmetry resulted				
> 40	122	114	550	51.69	18.15	No local symmetry resulted				
> 50	122	114	550	51.69	18.15	No local symmetry resulted				
> 60	122	114	550	51.69	18.15	No local symmetry resulted				
> 70	112	57	560	66.27	16.67	No local symmetry resulted				
> 80	90	0	582	100.00	13.39	No local symmetry resulted				
> 90	No local symmetry resulted					No local symmetry resulted				

Section 9.2.3) To avoid any data manipulation, we decided to present the original genuine experiment with current annotations.

Regarding the number of false negatives in ArgoUML (*cf.* Table 5), a detailed analysis showed that 17% of its unused features traces usually refer to the statements within the initialization classes, such as `Main` classes, or use other external libraries. These traces have not been used for the mapping due to the fact that *symfinder* does not categorize initialization classes as part of local symmetries and filters out external libraries.

Moreover, this average precision of 47.16% in *symfinder* is still acceptable, because the recall value show (91.74%, on the average) shows that *symfinder* identifies almost all expected variability in a system.

8.3 Distinguishing actual vp-s with variants

The resulted precision from the two subject systems shows that about half of the identified local symmetries by *symfinder* are actual *vp*-s with variants, while the other half are without a mapping, or irrelevant, to domain features. Since all identified local symmetries of a system are part of the same visualization, we need some means to distinguish the local symmetries that are actual *vp*-s with variants from the irrelevant local symmetries in visualization.

Our approach suggests that zones with the highest density of local symmetries in visualization contain the highest number of actual *vp*-s with variants (*cf.* Section 4.2). Hence, we expect that the irrelevant local symmetries belong to less dense places. To demonstrate it, we conducted the following experiment.

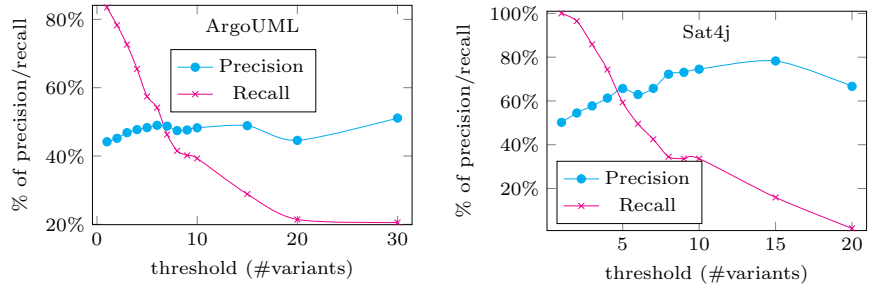


Fig. 14: Measurements based on density thresholds in ArgoUML and Sat4j

8.3.1 Experiment with the density threshold

We decided to introduce in *symfinder* a *density threshold* of local symmetries so that only zones with a higher density than the set threshold can be filtered in. We then observed how precision and recall vary for different density thresholds. For example, if the density threshold is set to 5 then only those local symmetries that are *vp*-s with more than 5 variants or other related *vp*-s at class or method levels will be used to calculate precision and recall. Hence, we aim to observe where are situated in the visualization most of the actual *vp*-s with variants, so they could be easily distinguished.

Technically, we just added a configuration option to the *symfinder*'s engine where a density threshold value could be set. Then, *symfinder* automatically identifies zones of higher density by adding a `HOTSPOT` label to local symmetries that are candidate *vp*-s with a higher number than the threshold of class or method level variants or other related candidate *vp*-s. These local symmetries are then filtered in and mapped to domain features using pre-existing features traces, such as in the cases of ArgoUML and Sat4j. As output, *symfinder* provides the calculated precision and recall for the set density threshold.

8.3.2 Results depending on the density threshold

In Table 6 are given the obtained results of precision and recall, using ArgoUML and Sat4j, for different density threshold values. The first row shows the results without a density threshold, which are the same as those in Table 5. We set a continuous range of density threshold values between 2 and 10, then we increased it by 5 or 10 until no local symmetry appeared.

From the results in Table 6, Figure 14 shows how precision and recall have changed depending on the density threshold. In both subject systems, by increasing the density threshold up to a specific value, precision is continuously improved because true positives (TP) decrease slower than false positives (FP) while recall is worsened because false negatives (FN) are increased. Concretely, zones with a density threshold higher than 6 and 5 in ArgoUML and Sat4j, respectively, contain a high number of actual *vp*-s with variants. This indi-

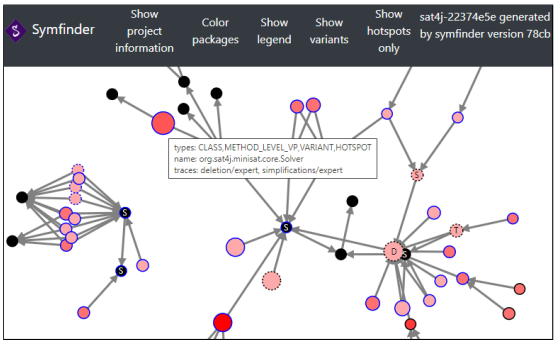


Fig. 15: An excerpt with highlighted actual *vp*-s with variants in Sat4j

cates that over half of actual *vp*-s with variants are situated into those zones that have more than 6 or 5 density of local symmetries. Put otherwise, as false positives (FP) decrease faster than true positives (TP) for the density threshold values up to 6 or 5, then most of the irrelevant local symmetries are situated into zones with a smaller density than 6 or 5, respectively. For higher density thresholds, precision and recall fluctuate because true positives (TP) with false positives (FP) decrease and false negatives (FN) increases distinctly.

Furthermore, precision and recall in both subjects for up to 6 or 5 density thresholds shows that in ArgoUML precision is improved by 4.82% and recall is worsened by 29.31%, whereas in Sat4j precision is improved by 15.47% and recall is worsened by 40.71%. In both cases, precision is improved slower than recall is worsened. This indicates that although most of the irrelevant local symmetries are in less dense zones, still these zones hold a considerable number of local symmetries that are actual *vp*-s with variants. Thus, actual *vp*-s with variants are spread in all zones with different density of local symmetries, but they are slightly more concentrated into zones with higher than 6 or 5 density.

These results show that differentiating actual *vp*-s with variants from irrelevant local symmetries in the current visualization is challenging. Therefore, whenever features traces are available, we extended the *symfinder*'s visualization by labeling and highlighting local symmetries that are actual *vp*-s with variants. For example, almost the same visualization excerpt as in Figure 11 (right) is given in Figure 15 from Sat4j. Unlike in the first visualization, nodes with a blue border in Figure 15 show those local symmetries that have a mapping to domain features, that is, the actual *vp*-s with variants. Also, hovering over a node that is an actual *vp* or variant, the `traces` label will have as value the feature name. For example, variant `Solver` in Figure 15 has two traces to the `Deletion` and `Simplification` features.

8.4 Summarized answer to RQ2

As an answer to the second research question (RQ_2), the calculated *precision* and *recall* on the ArgoUML and Sat4j systems show that about half of the identified local symmetries are actual *vp*-s with variants (44.17% and 50.22%, respectively) while they implement a high percentage (83.48% and 100%, respectively) of all given domain features. Based on our observations, those without a mapping could be because the available domain features are of coarse grain, the ground truth may be incomplete with features, mandatory features are sidestepped in the ground truth, or some of the identified local symmetries are simply not variability related.

Thus, to a great extent, the identified local symmetries by *symfinder* are actually *vp*-s with variants and implement up to 100% domain features. What becomes challenging is distinguishing them in the visualization from the irrelevant local symmetries. Therefore, we extended the experiment and provided evidence that zones with a higher density than 5 or 6 of local symmetries contain most of the actual *vp*-s with variants. Although this reveals a need for further study, notably to devise whether there is a general common threshold that could be reused among systems, we believe the results still show the suitability of our proposed visualization, based on the density of local symmetries. Besides, these results show that *symfinder* can improve variability management of a system by making explicit its expected variability to software architects through identifying and providing means to comprehend it.

9 An experience report

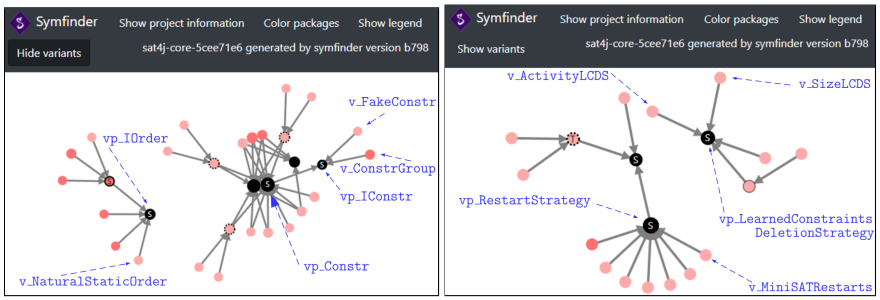
Towards addressing the third research question, RQ_3, we present here an experience report on the use of *symfinder* in the Sat4j system by its software architect.

9.1 Experimental setup

As opposed to the previous experiments, we decided to make an experiment with a software architect, asking him to use *symfinder* for comprehending the variability in his own variability-rich system and share his experiences. For this experiment we selected Sat4j [51], an internal variability-rich system.Its software architect, Daniel Le Berre [24], made the observations reported here and also helped improving the tooled approach, making him a co-author of this article.

In complement to reasons exposed in Section 6.2, we also chose Sat4j as it is a research software also implemented as an example in teaching software engineering. As such, it is designed according to good practices of object-oriented

[24] http://www.cril.univ-artois.fr/~leberre/



a: Identified *vp*-s with variants for features `Heuristics` with `Constraints` in Sat4j

b: Identified *vp*-s with variants for features `Restarts` with `Constraints database management` in Sat4j

Fig. 16: The identified *vp*-s with variants for four of the features in Sat4j

programming, including the good use of inheritance and design patterns to realize its variability, and the uniform usage of interfaces to prevent fragility. Its design has evolved over 15 years, mostly by enhancing its features and adding new ones.

The source code of Sat4j is divided into four modules: `core`, `pb`, `sat`, and `maxsat`. We used *symfinder* to identify the variability of only the `core` module, which contains the main features of the system. The generated visualization was then made available to the software architect.

9.2 Observations

We report here the different observations made by the architect while comprehending the identified variability by *symfinder* in Sat4j. He provided feedback on the general interest of *symfinder* for a software architect and on his particular interest for Sat4j. He also formulated requests for enhancements of the tooled approach.

9.2.1 Variability correctly identified by symfinder

Sat4j is a library of fully customizable Boolean solvers. Most features of a modern SAT solver are *variable and literal heuristics*, *restarts*, *constraints database management*, as well as the ability to handle various types of *constraints*. The feature model with its all features is given in Figure 13. In Sat4j, those features are configurable using the strategy design pattern. A solver solves by default a decision problem: several decorators are proposed to solve instead optimization problems. Finally, prebuilt solver configurations are made available through factories.

Most of the domain variability implemented using the strategy design pattern has been identified by *symfinder*. For instance, the *heuristics* are provided by the `IOrder` and `PhaseSelectionStrategy` interfaces, the *restarts* using

the `RestartStrategy` interface, the *constraints database management* with the `LearnedConstraintsDeletionStrategy` interface, the *constraints* with the `IConstr` interface. Their visualization is shown in Figure 16.

Note that Sat4j has two levels of abstraction: one for Java developers not familiar with the design of SAT solvers, and one for people with a deeper understanding of the algorithms (master students, researchers). Such abstractions can be seen with *symfinder* as the inheritance between two interfaces, see *e.g.*, `IConstr` and `Constr` in Figure 16a.

9.2.2 Variability missed by symfinder

There are only two *vp*-s that *symfinder* could not retrieve.

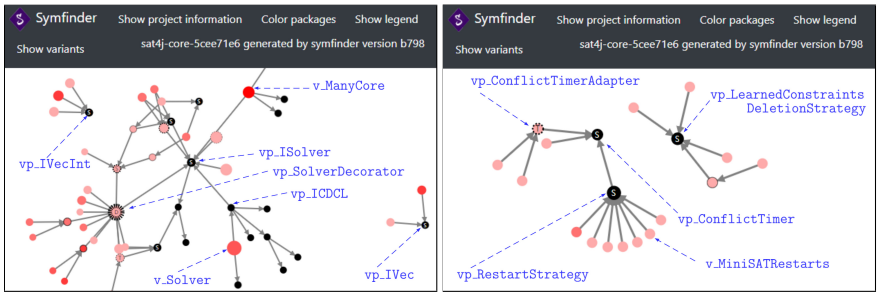
First, the interface `ISimplifier` and its implementations providing various clauses simplifications techniques were not detected. It is a specific tricky case as each implementation is an anonymous inner class inside the solver class, direct access to the state of the solver being required.

The second one is `PrimeImplicantStrategy`, which allows to reduce the model found by the solver to a set of literals required to satisfy all the constraints. Since that feature is experimental (this is the reason why it does not appear on Sat4j's feature model), the concrete implementation is chosen only if a model is found, from the value of a system property, if that computation is requested by the user. Then the object is used directly to perform the computation. Thus the interface is only found as a local variable in a method, not as a field in the `Solver` class.

In both cases, the variability is implemented in a very particular way, for either efficiency or limited scope reasons.

9.2.3 On the remaining identified variability

Some basic data structures like vectors with constant time operations not preserving the order of the elements (`IVec` and `IVecInt`) are identified as *vp*-s. They are not domain *vp*-s but implementation *vp*-s resulting from developers coding style or practices. `IVec` has two subclasses, shown in Figure 17a: `Vec` the concrete implementation and `ReadOnlyVec` a decorator preventing the modification of the enclosed `IVec`. Similarly, `IVecInt` has three concrete classes: `VecInt` the concrete implementation, `ReadOnlyVecInt` the decorator, and `EmptyVecInt` a null object design pattern. Most types manipulated in Sat4j are interfaces. In some cases, there is only one or two implementations of those interfaces. Being able to customize the minimal number of variants to consider for detecting a strategy (to 3 for instance instead of 2 in the current implementation) would allow to differentiate in this setting domain *vp*-s from implementation *vp*-s. All these findings also confirm that many false positive *vp*-s found in Sat4j were still variability implementations, but not related to the domain variability envisioned by the architect.



a: Identified *vp*-s with variants not related to domain features in Sat4j

b: Identified unexpected *vp*-s with variants in Sat4j

Fig. 17: The identified *vp*-s with variants not related to domain features and the unexpected ones

9.2.4 General interest of symfinder *for the software architect*

The first feedback that was reported is that the visualization enables him to quickly spot the main *vp*-s in the code. The way the nodes are grouped together is sometimes intriguing, different from the expected design. It may be the case that it evolved that way, or simply that the design has unexpected consequences. The most important feature of *symfinder* is to provide to the architect that global view to detect unexpected relationships and to be able to check the details of the nodes to decide if it is a design error or not.

Checking the variability in Sat4j was as easy as to look for interfaces marked as strategy and checking their name. Then the next steps consisted in checking abstract classes marked as strategy, then checking remaining plain interfaces (black node in the visualization) with several implementations, to make sure none is missed. Finally, like for each tool, one needs some time to get used to all the information displayed, and to learn the common patterns in the graphs.

9.2.5 Concrete interest of symfinder *for Sat4j*

symfinder identified the `ConflictTimer` interface as a strategy *vp*, while it is really an interface to implement a composite: timers are based on internal solver metrics instead of time to ensure reproducible results across platforms. A composite design pattern allows to trigger several events, seen as one from the solver perspective.

One can observe a double interface inheritance pattern for that interface in Figure 17b: `RestartStrategy` extends `ConflictTimer`. In that case, it does not correspond to the two levels of abstraction mentioned earlier. This is clearly a bad design choice, from a variability point of view. The reason of that inheritance is to allow the `RestartStrategy` variant to be added in the composite class of the `ConflictTimer`. Since all the variants have that requirement, that choice looks the simplest one from a developer perspective. Another choice

was done later for the `LearnedConstraintsDeletionStrategy` interface: each variant delegates to a specific timer, and this timer is added to the composite. There is certainly some refactoring work to do to uniformize the design of all *vp*-s.

Note however that if that inheritance link is removed, the `ConflictTimer` will still be detected as a strategy, since the interface has two concrete classes (due to the composite design pattern), thus satisfies the strategy detection conditions.

9.2.6 Requests for enhancement

Currently, *symfinder* hides some variants when they are all rooted to a common class. It happens a lot for Sat4j, since abstract classes are used to avoid as much as possible duplicated code in concrete classes. As such, most variants are not displayed. It does not allow for instance to identify quickly a strategy according to its number of concrete classes. This is important for the architect, since there is not that much textual information available by default.

It would also be nice to have a way to materialize Java packages. Most of the time, strategy interfaces are in the same package as the solver, while their implementation is in a dedicated package. It would be easy to quickly spot if all the variants of a strategy have been identified that way, or whether such practice is consistently used in the codebase [25].

On a more cosmetic point of view, it would help to have a specific color for each strategy interface and related concrete classes, to get an idea of the diversity of the *vp*-s.

Finally, the visualization could be complemented by the bulk list of all design patterns found, to validate them more easily. The graphical view is great to explore specific parts of the design or to highlight some zones of interests, but it is not necessarily convenient for exhaustive analysis.

9.3 Summarized answer to RQ3

The above observations point out that *symfinder* provides positive answers to all three research questions when applied to Sat4j. Even if the experiment was made on a single system, it especially answers RQ_3 and shows how *symfinder* can help in understanding the implemented variability. First, it can provide a global picture of the design of the variable software. Even if incomplete (only inheritance relationships are displayed, not delegation ones), such a broad picture allows to check that the variability appears at the expected places. This is possible because the number of incorrect classifications of *vp* is very low. Second, the visualization may allow to spot inconsistencies in the design, triggered by unexpected classifications. Again, this is possible because the number of false positives is low.

[25] This request was considered and is now addressed: `https://deathstar3.github.io/symfinder-demo/splc2020.html`

One potential improvement would be to help the architect to spot the missed *vp*-s, by providing *e.g.*, a list of candidate interfaces not classified as strategy. In Sat4j, those missing *vp*-s uncovered real design questions. It means that the value of the tool could be built both on detected and undetected variability.

A related question is "when in the development cycle *symfinder* could be useful?". In the particular case of Sat4j, it is clearly in the evolution phase: the expected and actual variabilities are unlikely to differ much at design time, because the code is written by a small team. However, when the code evolves, this is no longer true. The questionable inheritance relationship between `ConflictTimer` and `RestartStrategy` found by *symfinder* results from a new requirement from Eclipse three years after the initial design. In the general case, *symfinder* may also be useful at design time if the expected and actual variabilities may differ. However, it seems that it should be better used when the design is settled.

10 Discussions

In this section, we discuss the cross-checked results among the three experiments and threats to validity.

10.1 Cross-checking of results among three experiments for RQ1-RQ3

After conducting all three experiments, we cross-checked their results. Specifically, we checked if the obtained results for one research question can verify or help to further interpret the results for the two other questions.

First, the interpreted amount of variability in the sixteen subject systems in Section 7.2 is related to the obtained results on precision and recall of our tooled approach. Results on the second experiment in Section 8.4 with ArgoUML and Sat4j show that about half of the identified local symmetries are variability related, meaning that the rest may not be directly relevant to their respective domain features. Therefore, the interpreted amount of variability for all subject systems on the first experiment, given in Table 3, may require deeper insights about variability on those systems to lead to a better interpretation.

Then, contrary to the obtained data in Table 5 for Sat4j, the deep observations of its identified variability, given in the experience report in Section 9.3, show that it has very few false positive (FP) local symmetries. This suggests three possible deductions. First, it is an additional indication that some of the local symmetries in Sat4j given in Section 8.2 are falsely categorized as irrelevant because concrete classes are without annotations. Secondly, the large number of false positive local symmetries identified in ArgoUML and Sat4j could be related, to some degree, with the data normalization that we apply in order to be able to proceed with the experiment. Thirdly, as mentioned in Section 9.2.3, some very low level variability can be of no interest to architects.

All these cross-checked results may emphasize one interesting finding, that the identified variability in each system is specific and to some degree requires its own treatment and observations. Therefore, in the future, we aim to conduct similar observations by software architects in other subject systems to report and further cross-check the results.

10.2 Internal threats to validity

To address the RQ_2, we conducted an experiment using two real variability-rich systems, ArgoUML and Sat4j. While the ground truth of Sat4j is established by its software architect and features traces are part of its main branch, the ground truth of ArgoUML is established by a group of researchers. The main threat here is that another group of researchers or the ArgoUML's developers themselves may identify slightly different features and trace links. This will have a direct impact on the obtained results for precision and recall of our tooled approach calculated by ArgoUML.

Then, the software architect established the Sat4j's ground truth to address the RQ_2 after he reported the experience with *symfinder* to address the RQ_3. It seems as a maturation threat, but some basic knowledge for variability in his own system was essential to avoid adding meaningless annotations as features traces. Moreover, the reported experience with *symfinder* was conducted right after Sat4j's software architect was introduced to the *symfinder*'s approach.

During the data normalization of ArgoUML and Sat4j, we had to normalize the method level features traces and the identified local symmetries to class level. Including them in the experiment could have an impact on the obtained precision and recall. Hence, considering method level feature traces and local symmetries is inline with our future work.

10.3 External threats to validity

We only considered in our experiments Java-based variability-rich systems as that is the focus of the prototyped toolchain. Being able to analyse more languages would enable us to study more systems, but also projects architected with different languages, for example, with JavaScript for the front-end and Java for the back-end, such as in JHipster. Towards this, we just have extended the tool support for C++ variability-rich systems [69]. Therefore, we believe that our approach and toolchain can be extended to systems implemented by other languages and to other used techniques, including the implementation of variability at statement level by using the geometry of code [27, 13], for example, using line indentation [66].

Two of these subjects were also used to address the RQ_2. In both cases, we obtained approximately the same precision and recall for *symfinder* and also the same important density threshold. Despite this, the fact that there are only two systems limits the ability to draw general conclusions about all

other systems. For example, according to the visualization, we expect that the density threshold would be different in the case of Apache Maven 3.6.0, which has less variability implemented.

Finally, for addressing the RQ_3 we used Sat4j. It is the only system used in three experiments, therefore the ability to cross-check its results between the three experiments gives more validity to the third experiment. But, as a single analysed system in a qualitative experiment, and by one of the co-authors of this article, generalizing its results to other systems is unfeasible.

11 Related work

Reverse engineering SPL approaches. In the reengineering of *clone-and-own* and legacy software systems into SPL, there is a large body of work on feature location and feature identification approaches [6]. Feature location is an activity for automatically or manually recovering the traceability of some pre-existing features to the reusable code assets in an SPL [80, 23, 46]. Whereas, feature identification is an activity for identifying the common and varying units, as potential features, among some related software systems [101, 58]. In both cases, a set of clone-and-own or legacy systems are analysed. In contrast, we consider the class of object-oriented software systems that are variability-intensive but are not organized as a SPL. Then, instead of refactoring them into an SPL by identifying their domain features, for example, by doing an intersection of the abstract syntax tree elements of different systems, we automatically identify *vp*-s with variants, as two variability concepts that are close to code and abstract the implementation techniques or the reusable design of code assets. Regarding the classification of migration SPL engineering approaches [44], our variability identification process belongs more to reactive or incremental approaches. As the *symfinder* toolchain visualizes the identified variability implementations, we see more its usage to comprehend, and then refactor or incrementally extend the variability of a system under development.

Preprocessor-based approaches. Approaches for analyzing the variability of preprocessor-based systems seem more closely related to our work [53, 50, 37]. Similarly, we consider a family of systems within a single code base and study real software. Both approaches are likely to cover a large set of the most used variability implementation techniques in industrial settings. However, these works aim at comprehending the usage of C/C++ preprocessor directives for implementing variability, as a single technique, or at extracting them as features into a feature model. On our side, we provide some tool support for comprehending the variability of a software system implemented by a set of object-oriented techniques, including design patterns, without refactoring it into an SPL.

Variability visualization approaches. A recent mapping study shows that there are several approaches and tools for variability visualization, which mostly

came from information visualization in SPL engineering [54]. The most common visualized artifacts are feature models, which use trees or graphs. But, there are very few approaches for visualizing the variability at the code level. The existing ones use colors [43] or bar diagrams [24]. Some visualizations for feature-file tracing have also been proposed [3], but they are very specific. In general, excluding the configuration process [82, 75], it is well recognized that the majority of the tools in SPL engineering use *ad hoc* visualization techniques or the available functionalities inside Eclipse [54]. In contrast, our visualization tends to display, after filtering, trees – which are actually disconnected graphs – conform to the nature of *vp*-s, variants, and their relationships. Displaying classes, inheritance links, and some additional metrics, our visualization can be seen as related to the ones for understanding a large set of classes, such as polymetric views [48, 49]. However, the information we used is just focusing on local symmetries or the potential implemented variability, but relating other software metrics (e.g. quality metrics) to our set of information is clearly an interesting research topic. Toward that, relations and coupling can be studied with several advanced visualization techniques that are now used for software understanding, such as visualizing large codes as cities [95, 94], as hotspot maps, or as social networks [90].

Tools and prototypes. There is a large set of tools and prototypes for implementing or managing variability. Mostly they are developed in the context of SPL engineering, such as FeatureIDE [62] for forward engineering of SPLs. Then, the industrial variant management tool pure::variants [8, 32] provides also a variability management and visualization form for the realized variability into a *family model*, including code assets. Specifically, in the configuration editor they use a hierarchical "file explorer style", iconography for types of elements and "feature" states, and a matrix view. It is quite different from *symfinder*, as pure::variants has a larger scope, using a broader range of core assets in addition to the code assets, and is used especially during product derivation. But, similar to us, they also abstract the realized variability, well-known as a *family model*, the subject system being basically a single code-base variability system, usually referred to as a 150% model, and the *family model* being kept separated from the code assets.

12 Conclusion

Summary. Object-oriented software systems are more and more variability-intensive. They are developed to represent a family of systems within a single code-base, although not developed methodologically as a software product line. The variability in these systems is implicit and hardly documented as it is likely implemented using different traditional techniques (*e.g.*, inheritance, overloading, software design patterns). Still, it can be abstracted in terms of variation points with variants and their different properties.

In this paper, we proposed an identification approach that uses the property of local symmetry in software constructs to highlight and abstract different kinds of variation points with variants within a system in a unified way. We extended previous work on software symmetry [16, 15, 99, 100, 34, 98] to systematically map eight object-oriented software constructs, including four design patterns, to variability abstractions. Then, we reported on a prototyped toolchain, *symfinder*, that automatically identifies the corresponding candidate variation points with variants of a Java-based system, and provides a first form of visualization, which relies on the density of local symmetries to enable software architects to spot zones of interest *w.r.t.* variability. Besides, we conducted a threefold evaluation. First, we applied *symfinder* on sixteen large open-source systems. We used the number of candidate variation points with variants to gain insights regarding their variability. Secondly, we evaluated the precision and robustness of *symfinder* by measuring the number of candidate *vp*-s with variants that are relevant through mapping them to some preexisting domain features of two systems, ArgoUML and Sat4j. Finally, an experience report on the application of *symfinder* to Sat4j is provided by its software architect. The obtained results show that, *symfinder* can automatically identify the amount of candidate variability in real variability-rich systems (up to 13K *vp*-s with variants) and it is more robust (91.74%, on the average) than precise (47.16%, on the average). Then, along with some extensions, it can be particularly useful during the evolution phase of a variability-rich system.

We expect this contribution to be a concrete step towards a better comprehension and maintainability of variability implementation with traditional techniques, its documentation, and also a way to resume the discussion on how to implement and manage variability within the main decomposition of code.

Future work. In the future, we first plan to provide hints for comprehending the identified variability in a given software system by using the visualized density, visualized implementation techniques, and provided metrics and options in visualization by *symfinder*. Then, we aim to improve the scope of the toolchain regarding the identification of symmetry in other software constructs, being object-oriented or functional. The automation of navigation from the visualization to source code is also envisaged. We also plan to discern variability implementation patterns in large systems. For this reason, we aim at exploiting other software metrics [25]. Besides, the format of our identified *vp*-s with variants is in compliance with the *variability exchange language* (VEL) [72], a coming standard for exchanging the variability data among different variability management environments. Therefore, we plan to make available an export of identified *vp*-s with variants to the VEL format.

Declarations

Not applicable.

References

1. Alexander C (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. Alexander C, Carey S (1968) Subsymmetries. Perception & Psychophysics 4(2):73–77, DOI 10.3758/BF03209511
3. Andam B, Burger A, Berger T, Chaudron MR (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, ACM, VAMOS '17, pp 100–107, DOI 10.1145/3023956.3023967
4. Anquetil N, Kulesza U, Mitschke R, Moreira A, Royer JC, Rummler A, Sousa A (2010) A model-driven traceability framework for software product lines. Software & Systems Modeling 9(4):427–451, DOI 10.1007/s10270-009-0120-9
5. Apel S, Batory D, Kästner C, Saake G (2016) Feature-Oriented Software Product Lines. Springer
6. Assunção WK, Lopez-Herrejon RE, Linsbauer L, Vergilio SR, Egyed A (2017) Reengineering legacy applications into software product lines: A systematic mapping. Empirical Software Engineering 22(6):2972–3016, DOI 10.1007/s10664-017-9499-z
7. Bachmann F, Clements P (2005) Variability in software product lines. Tech. Rep. CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, URL `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7675`
8. Beuche D (2019) Industrial variant management with pure::variants. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B, ACM, pp 37–39, DOI 10.1145/3307630.3342391
9. Bosch J, Florijn G, Greefhorst D, Kuusela J, Obbink JH, Pohl K (2001) Variability issues in software product lines. In: International Workshop on Software Product-Family Engineering, Springer, PFE '01, pp 13–21, DOI 10.1007/3-540-47833-7_3
10. Caldiera VRBG, Rombach HD (1994) The goal question metric approach. Encyclopedia of Software Engineering pp 528–532, DOI 10.1002/0471028959.sof142
11. Capilla R, Bosch J, Kang KC, et al. (2013) Systems and software variability management. Concepts Tools and Experiences
12. Coplien J, Hoffman D, Weiss D (1998) Commonality and variability in software engineering. IEEE Software 15(6):37–45, DOI 10.1109/52.730836
13. Coplien JO (1998) Space: The final frontier. C++ Report 10(3):11–17
14. Coplien JO (1999) Multi-Paradigm Design for C++. Addison-Wesley Longman Publishing Co., Inc.
15. Coplien JO (2001) The future of language: Symmetry or broken symmetry? In: Proceedings of VS Live 2001, pp 1–7,

URL https://sites.google.com/a/gertrudandcope.com/info/Publications/Patterns/Symmetry/FutureOfLanguage
16. Coplien JO, Zhao L (2000) Symmetry breaking in software patterns. In: International Symposium on Generative and Component-Based Software Engineering, Springer, Springer, GCSE 2000, pp 37–54
17. Coplien JO, Zhao L (2020) Toward a general formal foundation of fesign. symmetry and broken symmetry. (Forthcoming publication)
18. Couto MV, Valente MT, Figueiredo E (2011) Extracting software product lines: A case study using conditional compilation. In: 2011 15th European Conference on Software Maintenance and Reengineering, IEEE, pp 191–200, DOI 10.1109/CSMR.2011.25
19. Cruz D, Figueiredo E, Martinez J (2019) A literature review and comparison of three feature location techniques using ArgoUML-SPL. In: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, ACM, VAMOS '19, pp 1–10, DOI 10.1145/3302333.3302343
20. Czarnecki K, Grünbacher P, Rabiser R, Schmid K, Wasowski A (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, ACM, VaMoS '12, pp 173–182, DOI 10.1145/2110147.2110167
21. De Lucia A, Deufemia V, Gravino C, Risi M (2010) Improving behavioral design pattern detection through model checking. In: 2010 14th European Conference on Software Maintenance and Reengineering, pp 176–185
22. Diehl S (2007) Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer Science & Business Media
23. Dit B, Revelle M, Gethers M, Poshyvanyk D (2013) Feature location in source code: A taxonomy and survey. Journal of Software: Evolution and Process 25(1):53–95, DOI 10.1002/smr.567
24. Duszynski S, Becker M (2012) Recovering variability information from the source code of similar software products. In: 2012 Third International Workshop on Product Line Approaches in Software Engineering, IEEE, PLEASE, pp 37–40, DOI 10.1109/PLEASE.2012.6229768
25. El-Sharkawy S, Yamagishi-Eichler N, Schmid K (2019) Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. Information and Software Technology 106:1–30, DOI 10.1016/j.infsof.2018.08.015
26. Fritsch C, Lehn A, Strohm T, Bosch R (2002) Evaluating variability implementation mechanisms. In: Proceedings of International Workshop on Product Line Engineering, sn, PLEES '02, pp 59–64
27. Gabriel RP (1996) Patterns of Software, vol 62. Oxford University Press New York
28. Gacek C, Anastasopoules M (2001) Implementing product line variabilities. In: Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context, ACM, SSR '01, pp 109–117, DOI 10.1145/375212.375269

29. Galster M (2019) Variability-intensive software systems: Product lines and beyond. In: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, ACM, VaMoS '19, pp 1–1, DOI 10.1145/3302333.3302336
30. Galster M, Weyns D, Tofan D, Michalik B, Avgeriou P (2013) Variability in software systems — a systematic literature review. IEEE Transactions on Software Engineering 40(3):282–306, DOI 10.1109/TSE.2013.56
31. Garcia J, Ivkovic I, Medvidovic N (2013) A comparative analysis of software architecture recovery techniques. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 486–496, DOI 10.1109/ASE.2013.6693106
32. pure-systems GmbH (2020) pure::variants. URL `https://www.pure-systems.com/products/pure-variants-9.html`
33. Halin A, Nuttinck A, Acher M, Devroey X, Perrouin G, Heymans P (2017) Yo variability! JHipster: A playground for web-apps analyses. In: Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems, ACM, VAMOS '17, p 44–51, DOI 10.1145/3023956.3023963
34. Henney K (2003) The good, the bad, and the koyaanisqatsi. In: Proceedings of the Second Nordic Pattern Languages of Programs Conference, VikingPLoP, vol 2003, pp 1–8
35. Heuzeroth D, Holl T, Hogstrom G, Lowe W (2003) Automatic design pattern detection. In: 11th IEEE International Workshop on Program Comprehension, 2003., pp 94–103
36. Hilliard R (2010) On representing variation. In: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ACM, ECSA '10, p 312–315, DOI 10.1145/1842752.1842810
37. Hunsen C, Zhang B, Siegmund J, Kästner C, Leßenich O, Becker M, Apel S (2016) Preprocessor-based variability in open-source and industrial software systems: An empirical study. Empirical Software Engineering 21(2):449–482, DOI 10.1007/s10664-015-9360-1
38. Jacobson I, Griss M, Jonsson P (1997) Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley Professional
39. Jayaraman K, Harvison D, Ganesh V, Kiezun A (2009) jFuzz: A concolic whitebox fuzzer for java. Proceedings of the First NASA Formal Methods Symposium pp 121–125, URL `https://ntrs.nasa.gov/search.jsp?R=20100024457`
40. John I, Lee J, Muthig D (2007) Separation of variability dimension and development dimension. In: Proocedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '07, pp 45–49
41. Kamali SR, Kasaei S, Lopez-Herrejon RE (2019) Answering the call of the wild? thoughts on the elusive quest for ecological validity in variability modeling. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B, ACM, SPLC '19, pp 143–150, DOI

10.1145/3307630.3342400
42. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst
43. Kästner C, Trujillo S, Apel S (2008) Visualizing software product line variabilities in source code. In: Proceedings of the 12th International Software Product Line Conference: 2nd International Workshop on Visualisation in Software Product Line Engineering, SPLC - ViSPLE '08, pp 303–312
44. Krueger CW (2001) Easing the transition to software mass customization. In: International Workshop on Software Product-Family Engineering, Springer, PFE '01, pp 282–293, DOI 10.1007/3-540-47833-7_25
45. Krüger J, Gu W, Shen H, Mukelabai M, Hebig R, Berger T (2018) Towards a better understanding of software features and their characteristics: A case study of marlin. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '18, pp 105–112, DOI 10.1145/3168365.3168371
46. Krüger J, Berger T, Leich T (2019) Features and how to find them: A survey of manual feature location. Software Engineering for Variability Intensive Systems pp 153–172
47. Krüger J, Mukelabai M, Gu W, Shen H, Hebig R, Berger T (2019) Where is my feature and what is it about? a case study on recovering feature facets. Journal of Systems and Software 152:239–253, DOI 10.1016/j.jss.2019.01.057
48. Lanza M, Ducasse S (2003) Polymetric views - a lightweight visual approach to reverse engineering. IEEE Transactions on Software Engineering 29(9):782–795, DOI 10.1109/TSE.2003.1232284
49. Lanza M, Ducasse S, Gall H, Pinzger M (2005) CodeCrawler: An information visualization tool for program comprehension. In: Proceedings of the 27th International Conference on Software Engineering, ACM, ICSE '05, pp 672–673, DOI 10.1145/1062455.1062602
50. Le DM, Lee H, Kang KC, Keun L (2013) Validating consistency between a feature model and its implementation. In: International Conference on Software Reuse, Springer, ICSR '13, pp 1–16, DOI 10.1007/978-3-642-38977-1_1
51. Le Berre D, Parrain A (2010) The sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation 7(2-3):59–64, DOI 10.3233/SAT190075
52. Le Berre D, Rapicault P (2009) Dependency management for the eclipse ecosystem: Eclipse p2, metadata and resolution. In: Proceedings of the 1st International Workshop on Open Component Ecosystems, IWOCE '09, pp 21–30, DOI 10.1145/1595800.1595805
53. Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (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, ICSE '10, pp 105–114, DOI

10.1145/1806799.1806819
54. Lopez-Herrejon RE, Illescas S, Egyed A (2018) A systematic mapping study of information visualization for software product line engineering. Journal of Software: Evolution and Process 30(2):e1912, DOI 10.1002/smr.1912
55. Lozano A (2011) An overview of techniques for detecting software variability concepts in source code. In: International Conference on Conceptual Modeling, Springer, ER '11, pp 141–150, DOI 10.1007/978-3-642-24574-9
56. Martinez J, Ziadi T, Bissyandé TF, Klein J, Traon YL (2016) Name suggestions during feature identification: the variclouds approach. In: Proceedings of the 20th International Systems and Software Product Line Conference, ACM, pp 119–123, DOI 10.1145/2934466.2934480
57. Martinez J, Assunção WK, Ziadi T (2017) Espla: A catalog of extractive spl adoption case studies. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume B, ACM, SPLC '17, pp 38–41, DOI 10.1145/3109729.3109748
58. Martinez J, Ziadi T, Bissyandé TF, Klein J, Le Traon Y (2017) Bottom-up technologies for reuse: Automated extractive adoption of software product lines. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, IEEE, ICSE-C '17, pp 67–70, DOI 10.1109/ICSE-C.2017.15
59. Martinez J, Ordoñez N, Tërnava Xh, Ziadi T, Aponte J, Figueiredo E, Valente MT (2018) Feature location benchmark with ArgoUML SPL. In: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, ACM, SPLC '18, pp 257–263, DOI 10.1145/3233027.3236402
60. Martinez J, Tërnava Xh, Ziadi T (2018) Software product line extraction from variability-rich systems: The robocode case study. In: Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1, ACM, SPLC '18, pp 132–142, DOI 10.1145/3233027.3233038
61. McKay BD, et al. (1981) Practical Graph Isomorphism. Department of Computer Science, Vanderbilt University Tennessee, USA
62. Meinicke J, Thüm T, Schröter R, Benduhn F, Leich T, Saake G (2017) Mastering Software Variability with FeatureIDE. Springer
63. Metzger A, Pohl K (2014) Software product line engineering and variability management: Achievements and challenges. In: Proceedings of the on Future of Software Engineering, ACM, FOSE '14, pp 70–84, DOI 10.1145/2593882.2593888
64. Metzger A, Pohl K, Heymans P, Schobbens PY, Saval G (2007) Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: 15th IEEE International Requirements Engineering Conference, IEEE, RE '07, pp 243–253, DOI 10.1109/RE.2007.61
65. Meyer B (1988) Object-Oriented Software Construction, vol 2. Prentice Hall New York

66. Miara RJ, Musselman JA, Navarro JA, Shneiderman B (1983) Program indentation and comprehensibility. Communications of the ACM 26(11):861–867
67. Michelon GK, Linsbauer L, Assunção WK, Egyed A (2019) Comparison-based feature location in ArgoUML variants. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A, ACM, SPLC '19, pp 93–97, DOI 10.1145/3336294.3342360
68. Mortara J, Tërnava Xh, Collet P (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, SPLC '19, Tools and Demonstrations, pp 5–8, DOI 10.1145/3307630.3342394
69. Mortara J, Collet P, Tërnava Xh (2020) Identifying and mapping implemented variabilities in java and c++ systems using symfinder. In: 24th International Systems and Software Product Line Conference, ACM, SPLC'20, DOI 10.1145/3382025.3414987
70. Mortara J, Tërnava Xh, Collet P (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, ACM, VaMoS '20, pp 1–9, DOI 10.1145/3377024.3377037
71. Niere J, Schäfer W, Wadsack JP, Wendehals L, Welsh J (2002) Towards pattern-based design recovery. In: Proceedings of the 24th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '02, p 338–348, DOI 10.1145/581339.581382
72. OASIS (2020) Oasis variability exchange language (vel) tc. URL `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=vel`, [Online; accessed 25-April-2020]
73. Paskevicius P, Damasevicius R, Štuikys V (2012) Change impact analysis of feature models. In: International Conference on Information and Software Technologies, Springer, ICIST '12, CCIS 319, pp 108–122, DOI 10.1007/978-3-642-33308-8_10
74. Patzke T, Muthig D (2002) Product line implementation technologies. programming language view. Tech. rep., Fraunhofer IESE
75. Pleuss A, Botterweck G (2012) Visualization of variability and configuration options. International Journal on Software Tools for Technology Transfer 14(5):497–510, DOI 10.1007/s10009-012-0252-z
76. Pohl K, Böckle G, van Der Linden FJ (2005) Software Product Line Engineering: Foundations, Principles and Techniques. Springer Science & Business Media
77. Rabiser R (2019) Feature modeling vs. decision modeling: History, comparison and perspectives. In: Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B, ACM, SPLC '19, pp 134–136, DOI 10.1145/3307630.3342399
78. Rosen J (1995) Symmetry in Science. Springer

79. Rosen J (2008) Symmetry Rules: How Science and Nature are Founded on Symmetry. Springer Science & Business Media
80. Rubin J, Chechik M (2013) A survey of feature location techniques. In: Domain Engineering: Product Lines, Languages, and Conceptual Models, Springer, pp 29–58, DOI 10.1007/978-3-642-36654-3-2
81. Schmid K, John I (2004) A customizable approach to full lifecycle variability management. Science of Computer Programming 53(3):259–284, DOI 10.1016/j.scico.2003.04.002
82. Schneeweiss D, Botterweck G (2010) Using flow maps to visualize product attributes during feature configuration. In: Proceedings of the 14th Software Product Lines Conference, Workshop Proceedings (Volume 2 : Workshops, Industrial Track, Doctoral Symposium, Demonstrations and Tools), Springer, SPLC '10, pp 219–228
83. Shi N, Olsson RA (2006) Reverse engineering of design patterns from java source code. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp 123–134
84. Shneiderman B (1996) The eyes have it: A task by data type taxonomy for information visualizations. In: Proceedings 1996 IEEE Symposium on Visual Languages, IEEE, pp 336–343, DOI 10.1109/VL.1996.545307
85. Stewart I, Golubitsky M (1992) Fearful Symmetry: Is God a Geometer? Courier Corporation
86. Svahnberg M, Van Gurp J, Bosch J (2005) A taxonomy of variability realization techniques. Software: Practice and experience 35(8):705–754, DOI 10.1002/spe.652
87. Tërnava Xh, Collet P (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, SPLC '17, pp 81–88, DOI 10.1145/3109729.3109733
88. Tërnava Xh, Collet P (2017) Tracing imperfectly modular variability in software product line implementation. In: International Conference on Software Reuse, Springer, ICSR '17, pp 112–120, DOI 10.1007/978-3-319-56856-0_8
89. Tërnava Xh, Mortara J, Collet P (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, pp 231–243, DOI 10.1145/3336294.3336311
90. Tornhill A (2015) Your Code as a Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs. Pragmatic Bookshelf
91. Turner CR, Fuggetta A, Lavazza L, Wolf AL (1999) A conceptual basis for feature engineering. Journal of Systems and Software 49(1):3–15, DOI 10.1016/S0164-1212(99)00062-X
92. Ullmann JR (1976) An algorithm for subgraph isomorphism. J ACM 23(1):31–42, DOI 10.1145/321921.321925
93. Vera-Pérez OL, Danglot B, Monperrus M, Baudry B (2019) A comprehensive study of pseudo-tested methods. Empirical Software Engineering

24(3):1195–1225, DOI 10.1007/s10664-018-9653-2
94. Wettel R, Lanza M (2007) Visualizing software systems as cities. In: 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, IEEE, pp 92–99, DOI 10.1109/VISSOF.2007.4290706
95. Wettel R, Lanza M (2008) Visual exploration of large-scale system evolution. In: 2008 15th Working Conference on Reverse Engineering, IEEE, pp 219–228, DOI 10.1109/WCRE.2008.55
96. Yu D, Zhang Y, Chen Z (2015) A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. Journal of Systems and Software 103:1 – 16, DOI https://doi.org/10.1016/j.jss.2015.01.019
97. Zhang B, Becker M, Patzke T, Sierszecki K, Savolainen JE (2013) Variability evolution and erosion in industrial product lines: A case study. In: Proceedings of the 17th International Software Product Line Conference, ACM, SPLC '13, pp 168–177, DOI 10.1145/2491627.2491645
98. Zhao L (2008) Patterns, symmetry, and symmetry breaking. Communications of the ACM 51(3):40–46, DOI 10.1145/1325555.1325564
99. Zhao L, Coplien J (2003) Understanding symmetry in object-oriented languages. Journal of Object Technology 2(5):123–134
100. Zhao L, Coplien JO (2002) Symmetry in class and type hierarchy. In: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, Australian Computer Society, Inc., ACM, CRPIT '02, pp 181–189, DOI 10.5555/564092.564119
101. Ziadi T, Frias L, da Silva MAA, Ziane M (2012) Feature identification from the source code of product variants. In: 2012 16th European Conference on Software Maintenance and Reengineering, IEEE, pp 417–422, DOI 10.1109/CSMR.2012.52