Towards Visualization of Feature Interactions in Software Product Lines

Sheny Illescas
Software System Engineering
Johannes Kepler University Linz
Austria
Email: k1257276@students.jku.at

Roberto E. Lopez-Herrejon
Dept. Software Engineering and IT
École de technologie supérieure, Canada
Email: roberto.lopez@etsmtl.ca

Alexander Egyed
Software System Engineering
Johannes Kepler University Linz
Austria
Email: alexander.egyed@jku.at

***Abstract*—Software Product Lines (SPLs) are families of related systems whose members are distinguished by the set of features they provide. To effectively evolve and maintain SPLs it is vital to understand how features are implemented and how they interact at different levels from source code to runtime. However the large number of features and the complex nature of interactions in typical SPLs make maintenance and evolution tasks challenging, and demand robust tool support for the software engineers to carry out these tasks. In this paper we present the first results of our ongoing work to address this need. We put forward four visualizations that focus on features and their interactions at source code level, evaluate them with four case studies, and sketch our future work.**

I. Introduction

Software Product Lines (SPLs) are families of related systems whose members are distinguished by the set of features they provide [3], [24], where a *feature* is an increment in program functionality [3]. *Variability* is the capacity of software artifacts to vary [28], and its effective management is a requisite for successful SPL development. Some of the benefits of applying SPLs are better customization, improved software reuse, and faster time to market [24], [30], [12].

Typical SPLs involve a large number of features and products (i.e. feature combinations) and consequently pose unique challenges to software engineers because the feature interactions in those products must be properly detected, analyzed, and managed. Broadly speaking, a feature interaction occurs when the behaviour of one feature changes depending on the presence or absence of another feature or set of features [14], [1]. Feature interactions manifest in a wide range of levels; for example, as source code artifacts, as unexpected executions, or as changes in non-functional properties. Research in feature interactions has a long standing history, and the interest in the subject has been rekindled in light of recent research developments in the context of SPLs [1].

However there is no robust support (e.g. no visualizations) for software engineers to identify, analyze and manage feature interactions which is indispensable for SPL performing evolution and maintenance tasks. In this paper we present the first results of our ongoing work to address this need. We propose four visualizations for features and feature interactions at source code level which we applied to four SPL case studies of different domains and sizes. We extend on our previous work on *Extraction and Composition for Clone-and-Own (ECCO)* which provides a framework for systematic support of SPL development that extracts and represents features, feature dependencies and feature interactions from legacy products [14], [9], [10]. This paper provides the first results on the feasibility of using visualization techniques for the ECCO framework and uncovers limitations and open issues.

The paper is structured as follows. Section II presents the necessary background on features, features interactions and our ECCO approach. In addition it introduces the running example that we use to illustrate ECCO's concepts and our visualizations. Section III presents and sketches the four visualizations we implemented. Section IV provides an overview of related work on SPL and visualization. Section V summarizes the conclusions obtained from this work and outlines our future work.

II. Background

In this section we provide the basic background and terminology necessary to understand our visualization proposals.

Features and Feature Interactions. In the area of SPLs there are many different conceptions of the term *feature* [5]. In our current work we focus on source code artifacts and take Zave's definition, which regards features as increments in program functionality [32], as our working definition. Similarly, there are also multiple conceptions and interpretations of the concept of feature interactions. Our focus is on those that happen at the source code level, referred to as *structural interactions* [1].

Running Example. We use as a running example an academic product line called *Drawing Product Line (DPL)* [9]. Each variant contains a combination of the following features: the ability to handle a drawing area (`DPL`), draw lines (`line`), draw rectangles (`rect`), select a color to draw with (`color`), fill the shapes drawn (`fill`) and clean the drawing area (`wipe`). Table I lists 16 variants of DPL. The selected features are denoted with ✓and the unselected features are left empty.

Figure 1 shows a code snippet that illustrates how DPL is implemented using preprocessor annotations, one of the most common techniques to realize SPLs [24]. The preprocessor annotations are highlighted in blue and mark which code parts are responsible for implementing which features. For example,

978-1-5090-3850-3/16 $31.00 © 2016 IEEE
DOI 10.1109/VISSOFT.2016.16

IEEE computer society

TABLE I: Products for DPL

Products	DPL	line	rect	wipe	color	fill
p1	✓	✓				
p2	✓	✓	✓			
p3	✓		✓			
p4	✓	✓	✓		✓	
p5	✓	✓			✓	
p6	✓		✓		✓	
p7	✓	✓		✓		
p8	✓	✓	✓	✓		
p9	✓		✓	✓		
p10	✓	✓	✓	✓	✓	
p11	✓	✓		✓	✓	
p12	✓		✓	✓	✓	
p13	✓	✓	✓		✓	✓
p14	✓		✓		✓	✓
p15	✓	✓	✓	✓	✓	✓
p16	✓		✓	✓	✓	✓

the field definition `List<Line> lines` shown in Line 3 will be included in class `Canvas` whenever the feature `LINE` is selected in a product, for example in product p1, p2, or p4.

As another example consider now the code in Lines 26 and 27. For this piece of code to be included in class `Line` of a product, the conditions in Line 23 and Line 25 must hold. This means that both features `LINE` and `COLOR` must be selected in such product. This is an example of a *structural interaction*, because this piece of code will be present in a product only when both features are selected and hence interact.

Another form of structural interaction is whenever the absence of a feature causes source code to be present, a phenomenon we refer to as *negative features* as we will shortly explain [14], [9]. As an example consider the code of Line 29 and Line 30, which shows the constructor of the class `Line` but this time with only one parameter. This piece of code is present in a product whenever feature `LINE` is selected but feature `COLOR` is not selected. Next we describe how our previous work provides a more formal footing to these ideas.

ECCO Approach. It is a common industry practice to develop new products by reusing artifacts from existing products in an ad hoc manner. This practice is referred to as *clone-and-own* and has documented maintenance and evolution problems [9]. In our previous work, we have proposed *Extraction and Composition for Clone-and-Own (ECCO)* to address the limitations of this practice by providing partially automated support for it [9], [10]. In this section we present how ECCO describes features and their interactions. For further details on ECCO's architecture, work flows, capabilities, and tool support please refer to [9], [10], [15].

ECCO describes structural features and their interactions by means of *modules* defined as follows [14], [9], [15].

Definition 1: A *module* is a set of signed features which are either positive (selected) or negative (unselected) that labels a set of artifacts.

ECCO modules can be of two types as defined next.

Definition 2: A *base module* labels artifacts that implement a given feature without any feature interactions, that is, it consists of exactly one positive feature and no negative features. We refer to them with the feature's name written in lowercase.

```
 1  class Canvas {
 2    #ifdef $LINE
 3    List<Line> lines;   // line
 4    #end
 5    #ifdef $RECT
 6    List<Rect> rects;    // rect
 7    #end
 8    #ifdef $COLOR
 9    Color color = Color.BLACK; // color
10    #end
11    ...
12    #ifdef $LINE
13    void mpLine(MouseEvent e) {    // line
14    ...
15    #ifdef $COLOR
16    // derivative δ^1(line,color)
17    newLine=new Line(color,start);
18    #end
19    ...
20    }
21  }
22
23  #ifdef $LINE
24  class Line {
25    #ifdef $COLOR
26    // derivative δ^1(line,color)
27    Line(Color c, Point start) {...}
28    #else
29    // derivative δ^1(line,¬color)
30    Line(Point start) {...}
31    #end
32    ...
33  }
34  #end
```

Fig. 1: DPL code example with annotations

For example, the field definition `List<Line> lines` shown in Line 3 will be included in class `Canvas` of *all* the variants that include feature `LINE`, independently of any other features being present. Hence this field definition is an *artifact* that is part of the base module `line`.

Definition 3: A *derivative module* $\delta^{n}(\mathtt{c}_0, \mathtt{c}_1, ..., \mathtt{c}_n) = \{c_0, c_1, ..., c_n\}$ labels artifacts that implement feature interactions, where c_i is `F` (if feature `F` is positive) or $\neg$`F` (if negative), and n is the order of the derivative. A derivative module contains at least one positive feature and any number of negative features.

If follows from the definition then that a derivative module of order n represents the interaction of $n+1$ features. Hence base modules have order 0, because they label a single feature and hence do no interact with any other features.

Let us now illustrate the concept of derivatives. Recall that the piece of code formed by Line 26 and Line 27 is included in a product when both features `LINE` and `COLOR` are selected. Hence, the artifact that represents these two lines belongs to derivative module $\delta^1(\mathtt{line}, \mathtt{color})$. Similarly, if in a product the feature `LINE` is selected but the feature `COLOR` is not selected, then Line 29 and Line 30 are included in class `Line`. Hence, the artifact that contains these two lines belong to derivative module $\delta^1(\mathtt{line}, \neg\mathtt{color})$. For a more detailed

explanation of the rationale behind negative features please refer to [14].

Definition 4: A dependency establishes a requirement relationship between two sets of modules and it is denoted with a three-tuple $(from, to, weight)$, where $from$ and to are a set of modules and $weight$ expresses the strength of the dependency (i.e. how many artifacts have the dependency).

In ECCO the dependencies denote the syntactic requirements of the artifacts at the source code level. An example is when a statement in an artifact belonging to a module makes a reference to a variable defined in another module. This is the case in Line 17 where variable $color$ is used in module $\delta^1(\texttt{line}, \texttt{color})$ but it is defined as part of module `color` in Line 9.

The ECCO tool computes all the modules and their dependencies based on the artifacts of the software products it receives as input and the combinations of features (e.g. see Table I for DPL). All the information is serialized into XML files (see [2], [10], [11]) that are used as input in our visualizations described next.

III. Visualization of ECCO modules

For the implementation of our visualization prototype we relied on Data-Driven Documents D3.js [8], [20], a Javascript library for manipulating documents based on data that uses HTML, SVG and CSS. D3.js provides methods for drawing visual representations of data including predefined templates. D3.js is a popular tool for data visualization because of the flexibility on what can be built from the provided components. We exploited this capability to develop our visualizations by adapting and extending available components.

We developed four visualizations that we applied to four SPLs whose main characteristics are summarized in Table II. `DPL` is our running example. `ZipMe` is a product line for a decomposed version of the Java-Pack-Library. `Video on Demand` (VOD) is a product line for video-on-demand streaming applications. `ArgoUML` is an open source project that has been made into a product line of UML Modeling tools. For further details on these case studies please refer to [2], [15]. We executed the following visualizations on a standard laptop computer with an Intel(R) Core i5-337U processor, CPU 1.80 GHz and 4 GB RAM.

TABLE II: Case Studies Overview

Case Study	NFeatures	NProducts	NModules	NArtifacts
DPL	6	16	336	630
ZipMe	7	32	940	37343
VOD	11	32	15520	34924
ArgoUML	11	256	52232	204418

NFeatures = number of features, NProducts = number of products, NModules = number of modules, NArtifacts = number of artifacts

A. Visualization of Dependencies

The first visualization focuses on the dependencies between modules and was implemented using D3's Hierarchical Edge Bundling template [6]. In this visualization the modules are sorted according to their respective order and the thickness of the connecting lines depends on the weight of the dependency. Figure 2 shows this visualization for DPL.

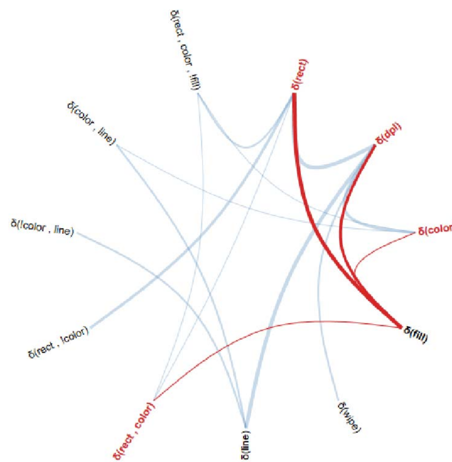

Fig. 2: Visualization of Dependencies of DPL

With this visualization the user can explore the relationships between the derivatives. For example, in Figure 2 the lines highlighted in red are the dependencies found from module `fill`, to modules `rect`, `dpl`, `color`, and $\delta^1(\texttt{rect}, \texttt{color})$. Notice that for visual simplicity all the modules are depicted with the δ symbol and without order number. The dependencies visualization it took to compute on average 1.3 ms.

B. Visualization of Derivatives

Derivatives are visualized using the Force-Directed Graph layout of D3.js as shown in Figure 3 for ZipMe SPL. The nodes with the prefix `!` correspond to the negative features, for instance the orange circle with the name `!gzip` at the bottom of the figure is the negation of the feature `GZIP`. This relationship is represented visually with the circles having the same color. The layout automatically calculates and sets an initial position for each circle but the user is able to change their position at any time. The lines connecting the circles represent the existence of a derivative containing the connected features. In cases where more than 2 features are present in the derivative, the lines connecting the features will have the same color. If the user puts the mouse over the line, the lines that are part of the selected derivative will be highlighted. Additionally, the name of the derivative will be shown to the user. This visualization took between 1.2 ms and 1.5 ms to compute.

C. Visualization of Modules

The third visualization depicts modules based on the Arc Diagram layout of D3.js. Figure 4 shows this visualization for VOD case study. In this visualization features are assigned a color in a palette. Circles represent features, that when depicted with a black border denote negative features. The user is able to select more than one feature and search for the

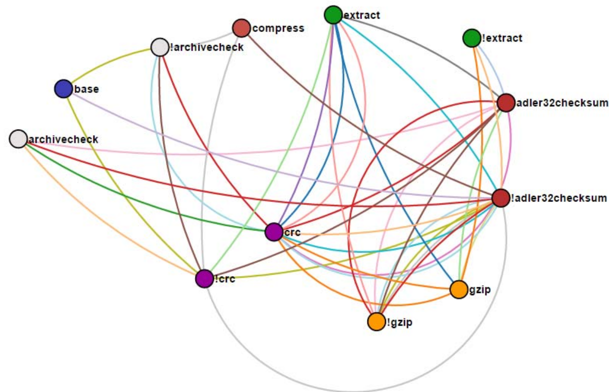

Fig. 3: Visualization of Derivatives ZipMe

modules in which they appear. In Figure 4 we select feature CHANGESERVER and searched for the modules containing this feature, yielding a total number of 5184 modules. The thickness of the lines represent the number of times the sequence containing the connecting features appears on the modules. The execution time depended heavily on the size of the case study. The execution times were DPL 894 ms, ZipMe 2.9 sec, VOD 9.8 sec and ArgoUML 10 min. These results shows an important scalability issue. For an elaboration on this finding and for further details please refer to [11].

D. Visualization of Artifact Trees

This visualization shows the structure of the artifacts, in our case studies these are Java AST nodes. ECCO stores artifacts as hierarchical trees that also contain cross-references, for instance when a statement calls a method [10], [15]. Our visualization relies on the *Pack Layout* of D3.js for representing artifact hierarchies. With this layout the size of each circle reveals a quantitative dimension of each data point [7]. The enclosing circles show the approximate cumulative size of each subtree. Figure 5 shows visualization for DPL case study. With this visualization the user can zoom in and zoom out through the different levels of the hierarchy by clicking on the artifacts. The circles filled in white represent the leaf nodes of the tree.

Circles with a dashed border indicate that the corresponding artifacts have references to other artifacts. If the user hovers

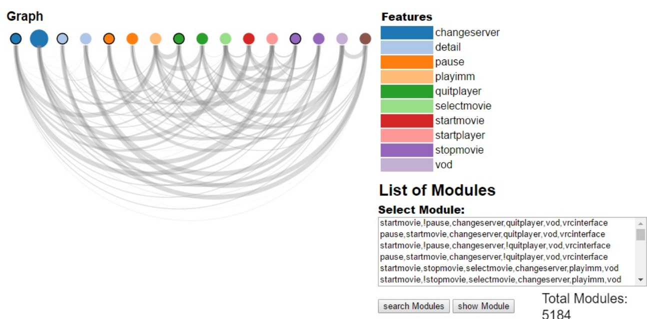

Fig. 4: Visualization of Modules of VOD with selected feature CHANGESERVER

the mouse over a circle with a dashed border, the circles representing the referred artifacts are highlighted. We found out that this approach proved useful for eliminating a large number of lines representing artifact cross-references.

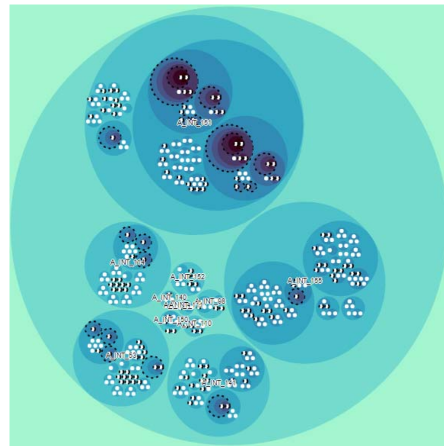

Fig. 5: Visualization of Artifacts of DPL

IV. Related Work

Software visualization is a subject which is gaining attention in the software engineering community. As some examples, Schots et al. performed an extensive study on software visualization for software reuse [26], [27], Novais et al. carried out a systematic mapping study in software evolution visualization [22], Prado et al. performed a systematic mapping study of visualization tools and techniques for software comprehension [25], and Paredes et al. carried out a systematic mapping study of the use of information visualization for software development following Agile approaches [23].

Visualization has also been applied to SPLs as attested by our recent work where we identified over 30 articles on this subject [17]. Next we briefly sketch some of our findings. In SPLs, *feature models* are tree-like structures that have become the de facto standard to visually represent the valid combinations of features [13], [4], which alternatively are captured as tables such as Table I for DPL. There have been several proposals to visualize feature models. For instance, Nestor et al. propose a tool for the interactive configuration of products based on feature models [21]. As another example, Urli et al. present a program comprehension approach called Variability Blueprint which is a polymetric-based visualization for representing the hierarchical structure of a feature model and its internal and external constraints [29]. Along the same lines Martinez et al. propose Feature Relations Graphs (FRoGs) as an interactive visualisation for domain experts and stakeholders to understand and maintain the constraints among features, and guide them through the creation of products [18]. Vasconcelos et al. also use feature models but in contrast for the selection of software visualizations [31].

There are also examples beyond visualization of feature models. For instance our previous work described also the

application of D3.js for the visualization in combinatorial interaction testing for SPL [16]. Murashkin et. al propose a visualization for multi-objective optimization of product configurations [19]

V. Conclusions and Future Work

In this paper we presented our ongoing work on visualization of features and their interactions at the source code level following the ECCO approach. We proposed four basic visualizations, that rely and extend components based on D3.js visualization tool, which we applied to four case studies of different domains and magnitudes. The implementation of our prototype and its applications let us gauge the potential benefits of our visualization for SPL developers but also uncovered limitations that we plan to address as part of our future work.

We found that as the amount of data to visualize increases, for instance as the number of features grows, can have important performance penalties in the visualizations. For example, zooming in and zooming out of the artifacts visualization is not instantaneous and it my require a couple of seconds for the large cases such as ArgoUML. In other to improve performance we are exploring options such as pre-processing the visualization data or capping the nesting levels of the artifacts. We also plan to extend our set of case studies to include SPLs implemented in languages other than Java, link our visualization with the source code editors, and integrate our prototype as a plugin in the ECCO tool.

VI. Acknowledgements

This research was partially funded by the Austrian Science Fund (FWF) projects P25289-N15 and P25513-N15.

References

[1] S. Apel, J. M. Atlee, L. Baresi, and P. Zave. Feature interactions: The next generation. Technical Report 14281, Dagstuhl Seminar, July 2014.

[2] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed. Extracting variability-safe feature models from source code dependencies in system variants. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, pages 1303–1310, 2015.

[3] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[4] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[5] T. Berger, D. Lettner, J. Rubin, P. Grnbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. What is a feature? a qualitative study of features in industrial software product lines. In *SPLC*, pages 16–25, 2015.

[6] M. Bostock. *Hierarchical Edge Bundling*, 2011. (accessed February 15, 2016).

[7] M. Bostock. *Pack Layout*, 2015. (accessed February 13, 2016).

[8] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[9] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 391–400, 2014.

[10] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. The ECCO tool: Extraction and composition for clone-and-own. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 665–668, 2015.

[11] S. Illescas. Visualization of variability realization. Technical report, Johannes Kepler University Linz, 2016.

[12] T. Käkölä and J. C. Dueñas, editors. *Software Product Lines - Research Issues in Engineering and Management*. Springer, 2006.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[14] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Recovering traceability between features and code in product variants. In T. Kishi, S. Jarzabek, and S. Gnesi, editors, *SPLC*, pages 131–140. ACM, 2013.

[15] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Variability extraction and modeling for product variants. *Software & Systems Modeling*, pages 1–21, 2016.

[16] R. E. Lopez-Herrejon and A. Egyed. Towards interactive visualization support for pairwise testing software product lines. In *2013 First IEEE Working Conference onSoftware Visualization (VISSOFT)*, pages 1–4. IEEE Computer Society, 2013.

[17] R. E. Lopez-Herrejon, S. Illescas, and A. Egyed. Visualization for software product lines: A systematic mapping study. In *VISSOFT*, 2016.

[18] J. Martinez, T. Ziadi, R. Mazo, T. F. Bissyandé, J. Klein, and Y. L. Traon. Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In *Second IEEE Working Conference on Software Visualization, VISSOFT 2014, Victoria, BC, Canada, September 29-30, 2014*, pages 50–59, 2014.

[19] A. Murashkin, M. Antkiewicz, D. Rayside, and K. Czarnecki. Visualization and exploration of optimal variants in product line engineering. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 111–115, New York, NY, USA, 2013. ACM.

[20] S. Murray. *Interactive data visualization for the Web*. " O'Reilly Media, Inc.", 2013.

[21] D. Nestor, S. Thiel, G. Botterweck, C. Cawley, and P. Healy. Applying visualisation techniques in software product lines. In *Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008*, pages 175–184, 2008.

[22] R. L. Novais, A. Torres, T. S. Mendes, M. G. Mendonça, and N. Zazworka. Software evolution visualization: A systematic mapping study. *Information & Software Technology*, 55(11):1860–1883, 2013.

[23] J. Paredes, C. Anslow, and F. Maurer. Information visualization for agile software development. In *Second IEEE Working Conference on Software Visualization, VISSOFT 2014, Victoria, BC, Canada, September 29-30, 2014*, pages 157–166, 2014.

[24] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[25] M. P. Prado, A. M. R. Vincenzi, F. A. A. de M. N. Soares, F. Cesar, G. P. de Paula, H. A. D. do Nascimento, J. C. Silva, J. L. de Oliveira, L. C. Lima, and T. Fernandes. Characterization of techniques and tools of visualization applied to software comprehension: A systematic mapping. In *International Conference on Software Engineering Advances (ICSEA)*, 2013.

[26] M. Schots. On the use of visualization for supporting software reuse. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 694–697. ACM, 2014.

[27] M. Schots, R. Vasconcelos, and C. Werner. A quasi-systematic review on software visualization approaches for software reuse. Technical report, Federal University of Rio de Janeiro, 2014.

[28] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Softw., Pract. Exper.*, 35(8):705–754, 2005.

[29] S. Urli, A. Bergel, M. Blay-Fornarino, P. Collet, and S. Mosser. A visual support for decomposing complex feature models. In *3rd IEEE Working Conference on Software Visualization, VISSOFT 2015, Bremen, Germany, September 27-28, 2015*, pages 76–85, 2015.

[30] F. J. van d. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[31] R. Vasconcelos, M. Schots, and C. Werner. An information visualization feature model for supporting the selection of software visualizations. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 122–125, 2014.

[32] P. Zave. Faq sheet on feature interaction. http://www.research.att.com/ pamela/faq.html.