VarMeR – A Variability Mechanisms Recommender for Software Artifacts

Iris Reinhartz-Berger and Anna Zamansky

Department of Information Systems, University of Haifa, Israel
iris@is.haifa.ac.il, annazam@is.haifa.ac.il

Abstract. Software is typically not developed from scratch and reuse of existing artifacts is a common practice. Consequently, variants of artifacts exist, challenging maintenance and future development. In this paper, we present a tool for identifying variants in object-oriented code artifacts (in Java) and guiding their systematic reuse. The tool, called VarMeR – a Variability Mechanisms Recommender, utilizes known variability mechanisms, which are techniques applied to adapt generic (reusable) artifacts to the context of particular products, for both identification of variants and recommendation on systematic reuse. Building on ontological foundations for representing variability of software behaviors, VarMeR visually presents the commonality and variability of the classes in different products and recommendations on suitable polymorphism variability mechanisms to increase systematic reuse.

Keywords: Software Product Line Engineering, Variability Analysis, Variability Mechanisms, Polymorphism, Ontology

1 Introduction

In practice, software reuse takes in many cases an ad-hoc form. Often while artifacts are not developed with reuse in mind, it is later achieved by duplicating artifacts and adapting them to the particular needs (a clone-and-own approach). Such an approach is easy to follow and intuitive, but has deficiencies in maintenance and future development. Thus, various methods have been suggested to detect variants, mainly in code, e.g., [1], [5]. Targeting at comparing and evaluating clone detection tools, four types of clones are mentioned in [2]: Type 1 – an exact copy without modifications (except for white space and comments); Type 2 – a syntactically identical copy (only variable, type, or function identifiers were changed); Type 3 – a copy with further modifications (statements were changed, added, or removed); Type 4 – a syntactically different copy which performs the same computation. Taking clone detection one step forward, a method, named ECCO (Extraction and Composition for Clone-and-Own), is introduced in [3] to enhance the clone-and-own approach with reuse capabilities. Given a selection of the desired features by the software engineer, ECCO finds the appropriate software artifacts to reuse and also provides hints whether they need adaptation. The adaptation itself, however, is left to the software engineer.

X. Franch J. Ralyté, R. Matulavi iųs, C. Salinesi, and R. Wieringa (Eds.):
CAiSE 2017 Forum and Doctoral Consortium Papers, pp. 57-64, 2017.
Copyright 2017 for this paper by its authors. Copying permitted for private and academic purposes.

In order to provide guidance to the adaptation process and to extract reusable artifacts which make future development and maintenance easier, we suggested in [11], [12], [13] a framework for identifying variants of software artifacts and associating them with variability mechanisms – techniques applied to adapt reusable artifacts to the context of particular products in Software Product Line Engineering (SPLE) [8]. The framework is based on ontological foundations, where software artifacts are viewed as things exhibiting behavior. The framework allows us to identify similar behaviors (rather than cloned realizations) and associate different variability mechanisms based on the characteristics of related similarity mappings. To support this approach, we have developed a tool called VarMeR – a Variability Mechanisms Recommender – which gets object-oriented code artifacts (in Java) that belong to two products and provides graphs that capture the commonality and variability of the classes of those products. The tool further recommends how to increase reuse by utilizing suitable variability mechanisms on similar classes. Currently VarMeR supports recommendations on three mechanisms related to polymorphism.

The rest of this paper is structured as follows. Section 2 provides the background of the approach, while Section 3 presents the capabilities of the VarMeR tool. Finally, Section 4 summarizes and refers to future development plans.

2 The Approach

The approach at the heart of VarMeR analyzes the commonality and variability of products behaviors and provides reuse recommendations in the form of associating polymorphism mechanisms to classes that behave similarly (even if their realizations are different). Accordingly, the approach is composed of three steps, which are shown in Figure 1 and elaborated next: Extract Behaviors, Compare Behaviors, and Analyze Variability.

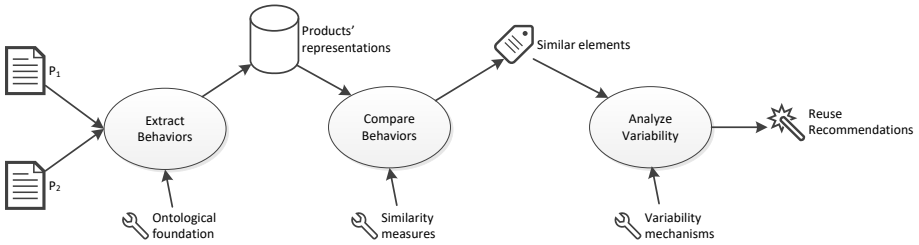

Figure 1. A high level overview of the approach

Extracting Behaviors. Referring to a software behavior as a triplet of initial state, external event, and final state [11], this step extracts those behavioral components from the operations of the different classes. Each class operation specifies some behavior of the software product. We assume that the operation name captures the essence of the behavior and thus can describe the external event, e.g., Borrow and Return of a Book Copy class in a library management system.

For extracting initial and final states, we distinguish between two levels: shallow – which refers to the signature of the operation, and deep – which takes into consideration the operation's behavior in terms of attributes used and modified throughout the operation[1]. The initial state of the behavior is composed of all the parameters passed to the operation (part of shallow) and all the class attributes used (read) by the operation (part of deep). The final state consists of the returned type (part of shallow) and all the class attributes modified (set) by the operation (part of deep). For the operation Borrow of the Book Copy class, we can consider the attributes AvailabilityStatus and BorrowingPeriod for the initial state, as they are needed for the operation to be executed. The attribute AvailabilityStatus is further modified as a result of the operation execution and hence is considered part of the operation's final state.

Compare Behaviors. After extracting the behaviors and their shallow and deep levels, a similarity mapping between their constituents is applied. This mapping can be based on existing general-purpose or domain-specific similarity metrics or some combination of such metrics. The metrics can take into account semantic considerations using semantic nets or statistical techniques to measure the distances among words and terms [10]. Alternatively, they can use type or schematic similarities, potentially ignoring the semantic roles or essence of the compared elements [6]. The similarity mapping associates to each operation's constituent (shallow or deep) all of its similar counterparts in the other operation (i.e., elements whose similarity with the given constituent exceeds some predefined threshold).

Analyze Variability. Suppose that the constituents of two operations o1 and o2 in classes C1 and C2 respectively are mapped using a similarity mapping *sim*, namely, the similarity of their constituents exceeded some predefined threshold. We can distinguish between the following situations with respect to *sim*:

1. USE – each constituent of o1 has exactly one counterpart in o2 and vice versa.
2. REF (abbreviation for refinement) – at least one constituent in o1 has more than one counterpart in o2.
3. EXT (abbreviation for extension) – at least one constituent in o1 has no counterpart in o2.

Note that REF and EXT are not mutually exclusive; we refer to a combination of both as REF-EXT (abbreviation for refined extension).

Aggregating the above notions from the level of operations to the level of classes, we aim at recommending on appropriate variability mechanisms. Our current focus is on the common polymorphism mechanisms. *Polymorphism* is the provision of a single interface to entities of different types. Therefore, the cases of polymorphism are characterized by similar signatures of operations (namely, the USE category in the shallow level of the operations). We further focus on three types of polymorphism which are widely used in industry [14]: subtype (inclusion) polymorphism (e.g., function pointers, inheritance), parametric polymorphism (e.g., C++ templates), and overloading. Table 1 presents recommendations for those polymorphism mechanisms based on the reuse mapping characteristics.

---

[1] We consider only attributes and ignore local variables, as the latter can be defined for implementation and realization purposes and may hinder the essence of the operation's behavior.

Table 1. Characteristics of Polymorphism Variability Mechanisms

Shallow	Deep	Description	Variability mechanism	Recommendation
USE	USE	Both signatures and behaviors are similar	Parametric polymorphism	Add complete behavior or behavior template as a core asset and utilize the parametric polymorphism
USE	REF	Signatures are similar and behavior is refined	Subtype polymorphism	Add the behavior as a core asset and utilize the subtype polymorphism
USE	EXT	Signatures are similar and behavior is extended	Subtype polymorphism	Add the behavior as a core asset and utilize the parametric polymorphism; use (procedure) calls to the less extended code
USE	REF-EXT	Signatures are similar and behavior is both refined and extended	Subtype polymorphism	As with Refinement and Extension
USE	Not mapped	Signatures are similar and behavior is different	Overloading	Add behavior interface as a core asset and utilize overloading

3 The VarMeR Tool

We implemented the approach in Java. The main inputs of the tool – VarMeR – are jar files implementing the software products (or applications) to be compared. Figure 2 presents the user interface of VarMeR: besides the names and paths to the compared jar files, the tool supports selection of similarity-related information, including thresholds, weights, and measures. Similarity measures define the way similarity is calculated. VarMeR currently supports the text semantic similarity metric of Mihalcea, Corley, and Strapparava (MCS) [10] that combines corpus-based and knowledge-based measures, the Latent Semantic Analysis (LSA) metric [9], and UMBC – top N similar words and phrase similarity metric [4]. The element and parameter name weights define the ratio between name and type similarities of elements (operations or attributes) and parameters, respectively. The weights are taken into consideration when comparing behaviors. As the names of parameters are more often meaningless (with respect to attribute/operation names), the tool supports separate weights. Finally, the similarity threshold defines the minimal value above which elements are considered similar.

The jar files of the compared products are reverse engineered into class diagrams (in XMI format) and Program Dependence Graphs (PDG)[2] [7] (in JSON format). The shallow and deep levels of the behaviors are extracted from those representations. Then the behaviors are compared utilizing the similarity-related information provided as input.

[2] PDG explicitly represents the data and control dependencies of a program.

Finally, variability is analyzed using the features of the three types of polymorphism (see Table 1).

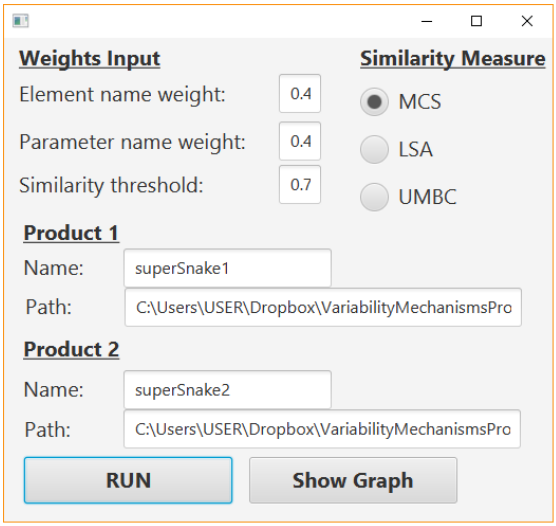

Figure 2. A screenshot of VarMeR inputs

The outcome of VarMeR is presented visually in the form of graphs. Each graph, comparing two products, shows the classes of the two products in different colors (one for the first product and one for the second product). The size of the nodes is proportional to the number of operations in the corresponding classes (the larger the node is, the more operations the class have). When hovering a node with the mouse, a tooltip showing the behavior appears, presenting the list of all operations of the current class. This way the user (e.g., a programmer or a code reviewer) can get a general idea on the role of each class, not just by its name, but also by the behavior it is expected to support.

The edges of the graphs (links between classes) represent recommendation on variability mechanisms, where:

- the label on the edge (link) indicates which variability mechanisms were identified: parametric, subtyping, and/or overloading.
- the width of the edge (link), as well as its length, represents the degree of evidence (i.e., the number of operations related with a certain type of polymorphism; the thicker/longer the link is, the more evidence to use the recommended variability mechanism exist).

An example of VarMeR's output is depicted in Figure 3. The numbers in parentheses in the tooltip indicate the numbers of operations with certain names (but different signatures). The top of the screen includes controls that allow defining the thresholds above which a given mechanism (parametric, subtyping, or overloading) will be presented. In other words, these weights separately control the minimal numbers of operation pairs that need to satisfy certain constraints (USE, REF, EXT, REF-EXT) so that the given mechanism will be recommended. In the top right part of the screen, the user can hide classes unrelated to classes in the other product (such as testFlight in Figure

3). The bottom of the screen supports selecting colors for the classes of each of the two products and the links between them. Note that currently, VarMeR compares classes from different products (and not classes from the same product that may further increase reuse). Hence, links connect nodes of alternating colors, potentially connecting a single node to several nodes representing classes in the other product (see scribeFilter in Figure 3 as an example).

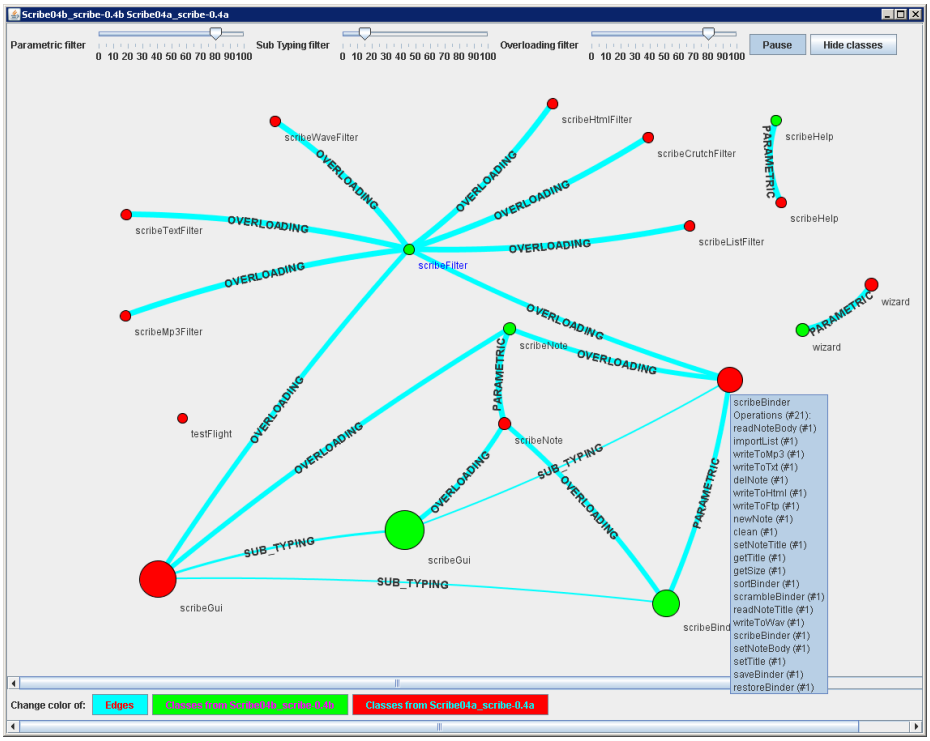

Figure 3. An example of VarMeR output

The tool further enables zooming into relations (links) among classes to have better understanding how the recommendations are generated and how to apply the recommended variability mechanisms and systemize reuse of the corresponding classes. This option presents the related operations of the classes and the categories to which the links among them belong: USE, REF, EXT, REF-EXT. Figure 4 zooms into the sub-typing link of Figure 3 presenting EXT and REF-EXT relations among operations of the corresponding classes. This mapping can be used by a programmer in order to create a class from which the two compared classes (ScribeGui and ScribeBinder) can inherit through sub-typing polymorphism.

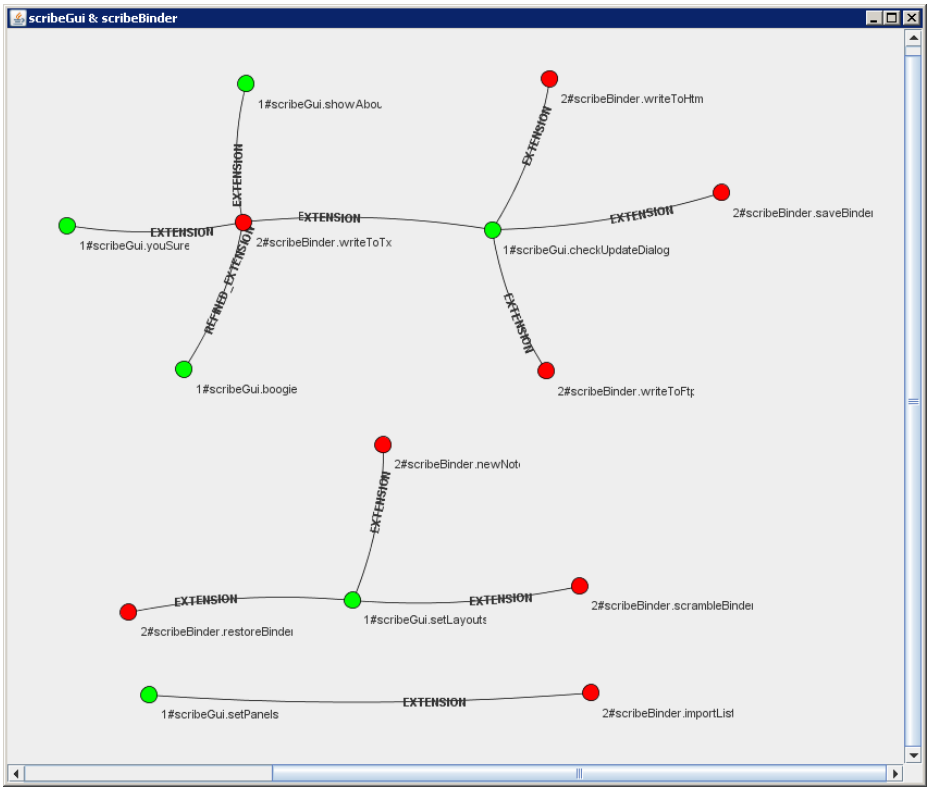

Figure 4. Zooming into a relation in VarMeR

4 Summary and Future Work

We presented a tool, called VarMeR – A Variability Mechanisms Recommender, which is based on ontological foundations for representing variability of behaviors of software products. The inputs of VarMeR are object-oriented code artifacts (jar files) belonging to different software products. The outputs are graphs which can be used for analyzing the commonality and variability of the classes in different products and recommending on suitable polymorphism variability mechanisms to increase systematic reuse.

In the future, we intend to extend the tool support in several ways. First, VarMeR will be extended to support further variability mechanisms besides polymorphisms. Another direction is incorporating VarMeR into existing programming environments, so that it will be relatively easy to apply the recommendations generated by VarMeR. Furthermore, we intend to empirically explore the usefulness of VarMeR and the quality of its outcomes.

Acknowledgment. The authors would like to thank Jonathan Liberman, Alex Kogan and Asaf Mor for their help in the implementation of the VarMeR tool. The second author was supported by the Israel Science Foundation under grant agreement 817/15.

References

[1] Baker, B. S. (2007). Finding Clones with Dup: Analysis of an Experiment. IEEE Transactions on Software Engineering 33 (9), pp. 608-621.

[2] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and Evaluation of Clone Detection Tools. IEEE Transactions on Software Engineering 33 (9), pp. 577-591.

[3] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. E., and Egyed, A. (2014). Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. IEEE International Conference on Software Maintenance and Evolution, pp. 391-400.

[4] Han, L., Kashyap, A., Finin, T., Mayfield, J., & Weese, J. (2013). UMBC EBIQUITY-CORE: Semantic textual similarity systems. In Proceedings of the Second Joint Conference on Lexical and Computational Semantics (Vol. 1), pp. 44-52.

[5] Kamiya, T., Kusumoto, S., and Inoue, K. (2002). CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering 28, pp. 654-670.

[6] Kashyap, V. and Sheth, A., (1996). Semantic and schematic similarities between database objects: a context-based approach. The VLDB Journal—The International Journal on Very Large Data Bases, 5(4), pp. 276-304.

[7] Krinke, J. (2001). Identifying Similar Code with Program Dependence Graphs. 8th Working Conference on Reverse Engineering, pp. 301-309.

[8] Lee, J. and Hwang, S. (2013). A Review on Variability Mechanisms for Product Lines. ICCA' 2013, ASTL vol. 24, pp. 1-4.

[9] Landauer, T. K., Foltz, P. W. and Laham, D. (1998). Introduction to Latent Semantic Analysis. Discourse Processes 25, pp. 259-284.

[10] Mihalcea, R., Corley, C., and Strapparava, C. (2006). Corpus-based and knowledge-based measures of text semantic similarity. American Association for Artificial Intelligence (AAAI'06), pp. 775-780.

[11] Reinhartz-Berger, I., Zamansky, A., & Wand, Y. (2016). An Ontological Approach for Identifying Software Variants: Specialization and Template Instantiation. 35th International Conference on Conceptual Modeling (ER'2016), pp. 98-112.

[12] Reinhartz-Berger, I., Zamansky, A., and Kemelman, M. (2015). Analyzing Variability of Cloned Artifacts: Formal Framework and Its Application to Requirements. Enterprise, Business-Process and Information Systems Modeling, EMMSAD'2015, pp. 311-325.

[13] Reinhartz-Berger, I., Zamansky, A., and Wand, Y. (2015). Taming Software Variability: Ontological Foundations of Variability Mechanisms. 34th International Conference on Conceptual Modeling (ER'2015), LNCS 9381, pp. 399-406.

[14] Zhang, B., Duszynski, S., and Becker, M. (2016). Variability Mechanisms and Lessons Learned in Practice. 1st International Workshop on Variability and Complexity in Software Design (VACE'2016), pp. 14-20.