Extracting Domain Behaviors through Multi-Criteria, Polymorphism-inspired Variability Analysis

*Iris Reinhartz-Berger and Sameh Abbas*

*Information Systems Department*

*University of Haifa, Israel*

*{iris@is.haifa.ac.il, samih1079@gmail.com}*

Abstract

Extraction of domain knowledge is an essential step towards developing new software systems and maintaining existing software products in the domain. Most current methods of extracting domain knowledge suppose high similarity of variants, which yields limited artifacts or low-level features that hide the domain behaviors. Our approach promotes a novel method for identifying domain behaviors in the form of a feature model; it starts by analyzing and detecting from low-level implementations, applies polymorphism-inspired mechanisms that are utilized by multi-criteria decision-making methods for producing candidate domain behaviors, then these are classified by machine learning techniques as local, global or irrelevant domain behaviors, and finally, dependencies are analyzed and a feature model is produced. The approach was evaluated on two datasets: one of the open-source video games, named apo-games, following a clone-and-own scenario; and variants of monopoly games, simulating a scenario of independent development of similarly behaving components.

1. Introduction and literature review

Concentrating on application families, the domain is a set of systems [1], that is, a composition of software components that were developed for a particular target [2]. The concept “domain” is also used in several contexts [3] besides application (system) sets; it is used in business, problem collections, and for the common terminology of a knowledge area. The domain describes shared properties, concepts, solutions, and behaviors. In a software context, analyzing the domain for extracting the common concepts and features is a labor-intensive task and error-prone. Previously, domain analysis depended on several domain “experts” who knew well the legacy systems and the domain of interest [2]. This mission, relying on experts, becomes difficult and almost impossible with the rise of the systems’ number and their increasing variability, so that a systematic approach becomes more essential.

The variability analysis of system families includes extracting the domain characteristics and features, as a core asset, and understating the optional variability. The extracted core asset can be reused as artifacts to develop a new system or to maintain existing systems. Establishing this issue systematically is known as Domain Engineering, namely, a systematic process for providing a common core architecture of system families in a manner facilitating their reuse for building a new system or maintaining an existing system of the domain [1].

The result of the domain analysis process is a *domain model*; the literature mentions several kinds of *domain model* production methods [1], where some of them are domain definition, context analysis, commonality analysis, domain lexicon creation, concept modeling, concept representation, and feature modeling. The widespread outcome of these is feature modeling [4], in which features are defined as prominent or distinctive user-visible aspects, qualities, or characteristics [4]. The features are commonly structured into trees or graphs, where the edges are dependencies of types “mandatory”, “optional”, “or” and “xor”. Cross-tree dependencies are also supported in the form of “requires” and “excludes” relations between features.

Feature modeling is the most widespread output of the variability analysis and domain exploring approach [4], and can be done by experts who know well the domain of interest of the system sets; however, with an increase in the number of systems, their complexity and their variability over time, automatic or semi-automatic extraction of the domain model and creation of the feature model become crucial. Systematically mapping the literature, Assunção et al. [5] observed a three-step process: (1) feature detection, mainly through feature location techniques [6]; (2) variability analysis, resulting in feature models; and (3) transformation, supporting the creation and implementation of core assets to be reused in the future development of systems in the domain.

The domain variability analysis process mostly relies on similarity metrics to explore the common, similar, or variant features. Many studies and promoted tools do that at the low level of implementation, especially, clone detection techniques [7] [8] based on the clone-and-own developing scenario. In a systematic review on clone detection, Ain et al. [8] classified six categories of clone detection approaches: (1) textual approaches; (2) lexical approaches; (3) tree-based approaches; (4) metric-based approaches; (5) semantic approaches; and (6) hybrid approaches. Clone detection approaches are suitable for systems developed by the same teams and for similar purposes, but for systems that are developed by different teams that still share similar behaviors the clone detection will not be practicable. Thus, analyzing the variability of these systems is necessary for developing new systems and maintaining existing systems.

Table 1 shows and compares several tools which were promoted for automatic or semi-automatic approaches and reviews variability analysis tools according to the Assunção et al. systematic mapping literature [5]. For each tool the analysis method can be expert-driven for a manual or a semi-manual process, static for an automatic process and dynamic for analyzing the systems during running time. Also, the table shows if the tool detects information from the input systems, like features or information about the variability and the commonality. The input artifacts can be in the source code of the programming language, requirements which can be in the specifications, feature descriptions, customer requests, test suites, documentation, design models such as class diagrams, state machines, and entity-relationship database models. Regarding the output, there are different types of artifacts that were generated; these were mostly feature mapped, namely, the features are given and the detecting process locates the relevant code for each feature, and then for the features discovered, it mainly extracts the feature’s elements. The reported artifacts generally represent variability information among the input systems. The last optional output is refactored source code that is proposed to be reused as an artifact for a software product line.

**Table 1,** Variability analysis tools according to Assunção et al. systematic mapping literature [5].

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| No. | Tool | Analysis | | | Detection | Input | | | Output | | | |
|  |  | Expert-driven | Static | Dynamic |  | Source code | Design model | Requirements | Features mapped | Features discovered | Reports | Source code refactored |
|  | Variability to Aspect tool |  | ✓ |  | ✓ | ✓ |  |  |  |  |  | ✓ |
|  | FeatureMapper | ✓ | ✓ |  | ✓ | ✓ | ✓ |  | ✓ |  |  |  |
|  | CoDEx Tool |  | ✓ |  | ✓ | ✓ |  | ✓ | ✓ |  |  |  |
|  | ThreeVaMar |  | ✓ |  | ✓ |  | ✓ |  |  | ✓ |  |  |
|  | Feature Model Extraction |  | ✓ |  | ✓ |  | ✓ |  |  | ✓ |  |  |
|  | RecFeat |  | ✓ |  | ✓ | ✓ |  | ✓ |  |  |  | ✓ |
|  | ETHOM |  |  |  |  |  | ✓ |  |  | ✓ |  |  |
|  | Clone-Differentiator Tool |  |  |  | ✓ | ✓ | ✓ | ✓ |  |  |  | ✓ |
|  | MapHist Tool |  | ✓ |  | ✓ |  |  | ✓ | ✓ |  |  |  |
|  | SPLevo tools |  | ✓ | ✓ | ✓ | ✓ |  |  |  |  |  | ✓ |
|  | Theme/SPL |  | ✓ |  | ✓ |  |  | ✓ |  | ✓ |  |  |
|  | BUT4Reuse | ✓ | ✓ |  | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
|  | ExtractorPL |  |  |  | ✓ | ✓ |  |  | ✓ |  |  |  |
|  | ECCO Tool | ✓ | ✓ |  | ✓ | ✓ |  |  | ✓ |  |  | ✓ |
|  | Model-Driven SaaS | ✓ | ✓ |  |  | ✓ | ✓ |  |  |  |  | ✓ |
|  | AUFM Suite | ✓ |  |  | ✓ |  |  |  |  |  | ✓ |  |
|  | JfeTkit |  | ✓ |  | ✓ | ✓ |  |  |  |  |  | ✓ |
|  | FMr-T |  |  |  | ✓ | ✓ |  |  |  | ✓ |  |  |
|  | ArborCraft | ✓ | ✓ |  | ✓ | ✓ |  | ✓ |  | ✓ |  |  |

Since the source code is the most widespread and available input and the most required output is the domain model, especially in the form of a feature model, needs to be done automatically. We therefore remain with three tools according to Table 1. The first is BUT4Reuse [9]; this tool extracts common blocks of code as a feature, and is appropriate for the scenario where one team has developed all the system sets implementing the clone-and-own method. Secondly, FMr-T [10] recovers feature models based on cloning. Thirdly, ArborCraft [11] extracts the feature model mainly based on the textual requirements document of the systems. All these three potential tools do not consider variability analysis based on recovering system behaviors that can be applied not only for systems that were developed by the same team, but also for systems that were developed by different teams but nevertheless behave similarly.

In this thesis, we promote an automatic variability analysis approach based on analyzing the functionality of a systems’ set, namely, extracting domain behaviors/operations that can be reused through a systematic process. Our approach proposes variability analysis that depends on the similarity metric we introduced in our former publication [12], which is applied by multi-criteria decision making (MCDM) based on polymorphism-inspired mechanisms [13], and is improved later by utilizing supervised machine learning [14] for automatic domain behavior extraction. Here we aim to put all previous approaches into a holistic domain behavior extraction approach, and to extend it to support the creation of a feature model for future reuse to develop new systems or to maintain existing systems in the domain.

The approach was evaluated using two different datasets. The first is called apo-games, which includes 20 video games that were suggested as variability challenges [15]; this data set represents the aspect of a set of applications (systems) that were developed by one team (actually one developer). The second dataset includes 17 monopoly games developed by different teams of software course students; this data set represents the aspect of a family of systems developed by different teams.

1. Motivation

One of the major gaps in the current literature is the absence of any variability analysis method, as most published research is based on detecting cloned blocks of code, which is reasonable for systems that were developed by one team using the clone-and-own method, but it is not sufficient for systems developed by different teams. In both cases the systems may share similar behaviors, and detecting the behaviors is important for discovering features and creating feature models. However, few works propose ways of detecting features and creating feature models; generally, they suggest feature location [6] for mapping a given feature to blocks of code, and few offer to create a feature model. Rather, these approaches are based on cloned code blocks with unclear or incomprehensible feature names.

1. Goals

The main goal of this work is to promote a systematic behavior variability analysis method for software system sets, whether developed by one team or different teams, in which the extracted domain model is based on detecting the functionalities of the system, namely, extracting the domain behavior in form of the feature model.

The RQs:

RQ1: How to detect domain behavior within the variability analysis?

RQ2: How will the feature model be built in accordance with the extracted domain behavior?

* 1. Methodology

In order to answer the RQs and achieve the goal, we derive a similarity comparator that can detect functionalities and behaviors from the source code of software systems. This is followed by filtering and verifying which extracted behaviors may be part of the final domain behaviors. Finally, we analyze the dependencies of the extracted domain behaviors and build a feature model.

To evaluate the suggested method, we use two datasets: the first is a family of software systems developed by the same development team, and the second is a set of software systems that were developed for the same goal by different development teams.

1. Suggested method

Our approach for domain behavior variability analysis consists of four main parts, as shown in Figure 1. The input is the source code of a family of systems, currently Java products. The first part (1) comprises parsing the input artifact, the operations descriptors that describe the operation’s signature (shallow descriptor), and the attributes that were used or were modified (deep descriptor). In the second part (2) similarity will be calculated among these operations by computing similarity between the shallow part of each couple operation, and between the deep descriptors of both, where the results are described by triple polymorphism-inspired mechanisms (*Parametric*, *Subtyping*, and *Overriding*). The *Parametric* mechanism resembles a high similarity (similar shallow, similar deep) tool, *Subtyping* is a moderate similarity (similar shallow, low deep similarity) tool, and *Overriding* is for when just the shallow descriptor is similar. The output is organized in a graph where the vertices are the product’s artifact; it may be operations at the low-level or higher-level classes, packages, or products. The edges are weighted by triple values (p, s, o) that describe the similarity in the forms of *Parametric, Subtyping,* and *Overloading*, respectively – PSO. Next as a similarity metric, we adopted MCDM (Multi-Criteria Decision Making) that is utilized for hierarchical clustering of similar artifacts using the triple similarity mechanisms PSO. The resulting clusters produced candidate domain behaviors. The previous results are identified and classified in the third step (3) by a supervised machine learning model, after gathering additional characteristics: size (number of instructions, methods, classes, and projects) and flow (invoked and invoked by), as well as the PSO for each candidate domain behavior. Finally (4) the dependency analysis step organizes the classified domain behavior into a feature model utilizing hierarchical clustering and set theory. The classification is done by supervised machine learning after training on a subset of apo-games, and depending on the manual feature model that is proposed in [16]. The classification distinguished between three types of domain behaviors: *local* domain behavior that is derived just from a single domain element (MCDM-aware cluster), *global* domain behavior when the domain behavior can be a combination that is derived from more than one domain element (MCDM-aware cluster), and *irrelevant* domain behavior, when there is no domain element that can be derived from it.

The evaluation was done by using two different datasets. The first is called apo-games which includes 20 video games that are suggested as variability challenges [15]. The second dataset includes 17 monopoly games developed by different teams of software course students. More details will be given in the Evaluation section.

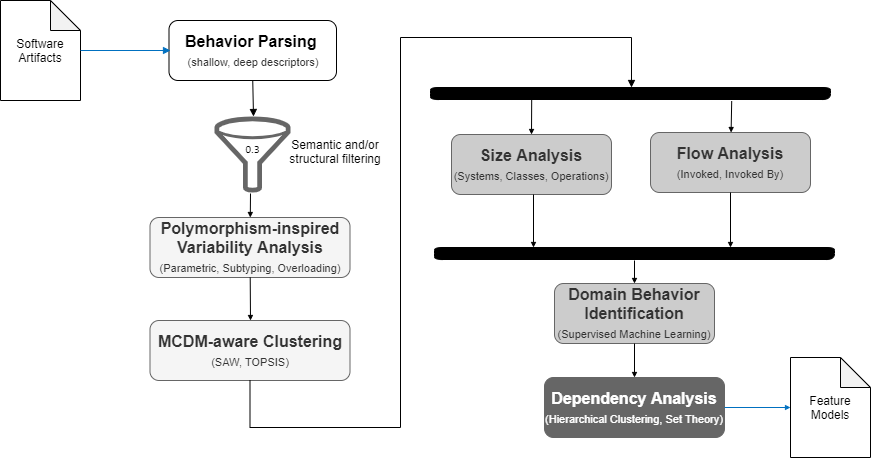


Figure 1 The suggested approach for domain behavior variability analysis

1. Evaluation

To evaluate our approach we utilized an extended version of VarMer [17], including all flow steps, as described in Figure 1. For the supervised machine learning stage, we organized a training dataset by classifying behaviors manually for 5 apo-game projects, and then selecting the best appropriate classifier with related optimum parameters for high accuracy. Next, we applied the fully automatic process to 20 apo-games and 17 monopoly games.

* 1. Datasets

The approach was evaluated using two datasets. The first is apo-games that was developed as a reverse engineering challenge [15] by a single experienced developer between the years 2006-2012. The challenge includes 20 open-source video games. Their development is characterized by using clone-and-own for reusing targets. The second set is 17 monopoly games that were developed by different student teams in a software engineering course. All teams had the same requirements and developed their game according to MVC (Model, View, Control); we concentrate our evaluation on the (M) Model part.

* 1. Evaluation Procedure

In order to prepare the approach for automatic execution we selected five apo-games for the training step to get the best classification model and its optimum specifications. Firstly, we ran the approach until MCDM-aware clustering (see Figure 1), which yielded 71 candidate domain behaviors. Then, we classified them manually according to the feature model that was created and presented for the same five games by different researchers for different purposes [16]. The feature model mainly represents concept elements, none of which is behavior, so that our classification to irrelevant, local, and global refer to these elements. Table 2 shows the classification report for the test set. We can see that the local and global classes get high precision and recall. The irrelevant type was not detected because of a its very low number, so we decided to deal with irrelevant classified domain behavior as global. Finally, we ran the full approach on the two data sets using the selected classifier.

Table 2. The classification reports.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Precision** | **Recall** | **F1-score** |
| **Local** | 0.86 | 0.86 | 0.86 |
| **Global** | 0.71 | 1.00 | 0.83 |
| **Irrelevant** | 0.00 | 0.00 | 0.00 |
| **Overall** | **0.79** | **0.79** | **0.79** |

* 1. Results

The approach detected 131 candidate domain behaviors in the apo-game dataset; 96 of them were classified as local domain behaviors, and were related to the games graphics elements handling, like button handling, image (buffering, updating, drawing), applet life-cycle (initializing, stopping, destroying), animations, and game management such as level (loading, generating, randomizing), the player (loading, making, drawing), high-score (loading, starting), and others. In addition, 21 domain behaviors were classified as global, referring to common game behaviors such as threads that were used for different purposes in the context of games, like animations, multi-tasking, and parallel playing. There also were behaviors common to button and mouse interactions, and behaviors for properties storage (I/O). The remaining 14 were classified as irrelevant, where some of them were very abstract, like adding, making, and selecting.

For the monopoly dataset, the approach produced 49 domain behaviors candidates, 40 of which were classified as local domain behaviors that represent the high-level monopoly game elements, such as users (admin and players), questions and answers, assets, jail, money, and tile. The remaining six global behaviors referred to rounds handling, data storage (I/O), and asset status handling. Three were classified as irrelevant and referred to behaviors for locating, answer checking, and system data handling.

Some explanation for the differences between the two-dataset output is that we analyze the model part of the monopoly projects (using the MVC – Model, View, Control programming pattern), neglecting the view and control parts. Another source of difference may be due to the clear well-defined requirements of a monopoly game, using the same high-level game concept; furthermore, it is a universally known game. On the other hand, the apo-games dataset represents the different games that were developed by a single programmer using a clone-and-own scenario and especially using specific abstract game packages, e.g. component and entity, that led to many mandatory domain behaviors. Another explanation is that we used for both datasets the same machine learning classifier that was built using a subset of the apo-games dataset.

.

* 1. Conclusions

The most challenging issues treated by our approach are the systematic process for extracting a feature model based on behavioral variability analysis, and the ability to extract these high-level features from low-level implementations that force us to improve our approach by cleaning very low-level behaviors such as getters, setters, to-String, and constructors.

The promising achievement is the extraction of a high-level abstraction as a reasonable feature model due to two main extensions: MCDM-aware hierarchical clustering that selects automatically the domain behavior candidates, and supervised machine learning for classifying each domain behavior candidate.

According to the results, we can see that the approach can deal with both programming scenarios: first, projects that were developed for the same matter by different teams (monopoly) and second, clone-and-own scenarios (apo-games).

Despite the threats of validity that can result from using a sub-set of apo-games to train the supervised machine learning model or from the deficiency of approaches that detect behaviors or systematically create feature models for these datasets, the approach nevertheless extracts the main behavior and related elements of the product's domain - maybe at different levels than has been achieved from the previous datasets.

1. Summary

In this thesis, we introduced an automatic holistic behavior variability analysis approach. The approach extracted a domain model in the form of a feature model from the low-level implementation of software system families, either developed by the same team or by different teams. This was achieved in four main stages: (1) parsing source code using the shallow (behavior interface) and deep (behavior transformation) descriptions to calculate the polymorphism-inspired variability mechanisms; (2) using these values with a MCDM method to cluster similar operations to be domain behavior candidates; (3) further classification into local, global, and irrelevant, utilizing supervised machine learning; and (4) dependency analysis for classified domain behaviors that are considered to be features which were organized by a feature model. The approach was implemented by VarMeR[17] tool and evaluated by two datasets from the games domain.

* 1. Contributions

Most of the literature promotes variability analysis, based on detecting cloned code blocks performed by the clone-and-own scenario that is used generally within the same developing team. Instead, our approach introduces variability analysis based on behavior extraction from the low-level source code of software system sets that can be developed by the same team or by different teams; it then creates a domain model in the form of the feature model. Here, we promote a method whose input is low-level source code and introduce a high-level feature model, where most current promotions for detecting feature model methods rely on high-level input. For example, input can be source code and given features from the design stage or proposed by experts, and the remaining mission is just to locate the code for each feature, namely, the mapping process.

The resultant feature model is considered to be the base of a core asset to create a reusing artifact that can be used for developing new systems or maintaining existing ones.

* 1. Threats to validity

Since we promote an innovative approach, there are threats of validity concerning the evaluation, the validity procedure, and the approach stages, outcomes, and selected approach parameters, as well as our use of third-party specific algorithms.

The first threat stems from our “behavior similarity” suggestion: there is no literature that suggests similarity tools to deal with this kind of similarity, which means that there is no such approach with which we can compare both processes and results. Despite this, we evaluated the approach using two datasets with different development scenarios, but we still need to investigate other scenarios.

The second threat derives from the fact that the feature model outcome cannot be compared to another approach outcome due to two reasons: first, our model represents “behavioral features” and there are no competing approaches for detecting behaviors; second, there is no even manual feature model for the two-evaluation datasets. Only for a subset of apo-games we found a very high-level manual feature model that we used for the manual classification while preparing the machine learning training set, but for the monopoly dataset, there is no manual or automatic feature model.

Thirdly, despite our process being automatic, we still need to prepare the machine learning stage individually, which means organizing a sufficient training set and selecting a better classifier and its optimum parameters. Therefore, we need domain expert involvement for manual classification and to select the optimum classifier.

A fourth threat, also related to the machine learning stage, is that there is no trusted professional training dataset that is specialized in domain behaviors that may be reliable for training machine learning. Currently, we embedded classifiers optimized on the training set that we prepared using a subset of apo-games, so it is specialized to the games domain. We also used it for the monopoly dataset, but perhaps it is not appropriate for this type of game. In all cases, we need to validate the relevance of this training set to other domains.

Fifth, our approach used many algorithms and methods in its different stages for optimizing parameters and third-party methods, for example, a semantic similarity method and related threshold, a MCDM method, and selected weights, polymorphism-inspired mechanism thresholds. All these parameters should be optimized systematically.

Finally, we evaluated the extraction of a feature model by conforming it in the context of the domain, according to the systematic mapping [5] domain reengineering essential for developing new software or marinating existing systems. Going forward, we need to evaluate how our extracted feature model contributes to this issue.

* 1. Future research

We now discuss some possible future directions related to (1) improving the current approach, (2) enhancing the evaluation method, (3) improving the current approach's performance, and (4) extending our approach to support other variability mechanisms and transforming the output towards SPL (software product line).

First, there are several possible directions for improving the current approach. We need to build a trusted training dataset to classify domain behaviors utilized by the machine learning stage, and maybe to support various domains. In addition, to derive an automatic process for selecting the best classifier and its optimum specifications. Another possible improvement of the clustering process is by examining other graph-clustering algorithms instead of the MCDM-SAW method to improve the performance. Related to variability analysis, an additional direction is extending the method to other well-known variability mechanisms, such as template instantiation and analogy.

Second, our evaluation confirmed the feature model output, but we need to evaluate the output quality and compare it to the actual feature model where software systems were developed according to it. We also need to assess how the feature models can support the maintenance of existing systems and the development of future ones in the given domain.

Third, we did not examine the approach performance at all the stages; in the future we intend to assess the performance for each stage and suggest ways for improvement.

Fourth and finally, with regard to moving forward towards SPL -- after the analyzing and detecting phases one must still perform the transformation stage, and suggest refactoring ways based on extracted feature model to produce a core asset for developing a new system and maintaining existing ones.

Acknowledgments

My most sincere thanks go to my supervisor, Prof. Iris Reinhartz-Berger. Her constructive criticism, profound knowledge, and patient guidance have contributed throughout the way.

References

[1] M. Harsu, *A survey on domain engineering*, vol. 12. Citeseer, 2002.

[2] W. Tracz, “DSSA (domain-specific software architecture) pedagogical example,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 20, no. 3, pp. 49–62, 1995.

[3] K. Schmid, “Scoping software product lines,” in *Software Product Lines*, Springer, 2000, pp. 513–532.

[4] K. Kang, S. Cohen, J. Hess, W. Novak, and A. S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study. Software Engineering Institute,” *Univ. Carnegie Mellon, Pittsburgh, Pennsylvania*, 1990.

[5] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, “Reengineering legacy applications into software product lines: a systematic mapping,” *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 2972–3016, 2017.

[6] J. Rubin and M. Chechik, “A survey of feature location techniques,” in *Domain Engineering*, Springer, 2013, pp. 29–58.

[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, Sep. 2007, doi: 10.1109/TSE.2007.70725.

[8] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, “A systematic review on code clone detection,” *IEEE access*, vol. 7, pp. 86121–86144, 2019.

[9] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, “Bottom-up adoption of software product lines: a generic and extensible approach,” in *Proceedings of the 19th International Conference on Software Product Line*, 2015, pp. 101–110.

[10] J. Maâzoun, N. Bouassida, and H. Ben-Abdallah, “Feature model recovery from product variants based on a cloning technique.,” in *SEKE*, 2014, pp. 431–436.

[11] N. Weston and A. Rashid, “ArborCraft: Automatic feature models from textual requirements documents,” in *Proceedings of the 15th workshop on Early aspects*, 2009, pp. 45–46.

[12] I. Reinhartz-Berger and S. Abbas, “Extracting domain behaviors through multi-criteria, polymorphism-inspired variability analysis,” *Inf. Syst.*, p. 101882, 2021.

[13] I. Reinhartz-Berger and A. Zamansky, “Reuse of Similarly Behaving Software through Polymorphism-Inspired Variability Mechanisms,” *IEEE Trans. Softw. Eng.*, 2020, doi: 10.1109/TSE.2020.3001512.

[14] I. Reinhartz-Berger, S. Abbas, and A. Zamansky, “A Variability-Driven Analysis Method for Automatic Extraction of Domain Behaviors,” in *CAiSE 2020*, 2020.

[15] J. Krüger, W. Fenske, T. Thüm, D. Aporius, G. Saake, and T. Leich, “Apo-games-a case study for reverse engineering variability from cloned Java variants,” in *ACM International Conference Proceeding Series*, 2018, vol. 1, pp. 251–256, doi: 10.1145/3233027.3236403.

[16] J. Debbiche, O. Lignell, J. Krüger, and T. Berger, “Migrating Java-based apo-games into a composition-based software product line,” in *ACM International Conference Proceeding Series*, 2019, vol. A, pp. 1–5, doi: 10.1145/3336294.3342361.

[17] I. Reinhartz-Berger and A. Zamansky, “VarMeR-A Variability Mechanisms Recommender for Software Artifacts.,” in *CAiSE-Forum-DC*, 2017, pp. 57–64.