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 yield limited artifact or low-level feature that hide the domain behaviors. Our approach promotes a novel method for identifying domain behaviors in form of a feature model, starts analyzing and detecting from low-level implementations, applies polymorphism-inspired mechanisms that are utilized by multi-criteria decision-making methods for producing candidate’s domain behaviors, then they are classified by machine learning techniques as local, global or irrelevant domain behavior, finally, dependencies will be analyzed and produced a feature model. The approach was evaluated on two datasets: one of the open-source video games, named apo-games, following a clone-and-own scenario; and the other are 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], a composition of software components that were developed for a particular target[2]. The concept “domain” is also used in several scopes [3] besides applications (systems) set, it is used in business, problem collections, and used 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 analyzing depended on several domain “experts” who know well the legacy systems and the domain of interest [2]. This mission, relying on experts, becomes difficult and almost impossible with the growth of the systems’ number, and increasing the variability of them, so, the systematic approach becomes more essential.

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

The result of the domain analysis process is *domain model*, the literature mentioned several kinds of *domain model* production [1] some of them are domain definition, context analysis, commonality analysis, domain lexicon, concept modeling, concept representation, and feature modeling. The widespread outcome of them is the feature modeling [4]. in which features, 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 for, of “requires” and “excludes” relations between features.

Feature modeling is the most widespread output of the variability analysis and domain exploring approach [4], where can be done by experts who know well the domain of interest of the systems set, but with increasing the number of systems, their complexity and variability over the time, the automatic or semi-automatic extracting domain model and creating feature model becomes 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.

Analyzing domain variability 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 the 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, (6)and hybrid approaches. Clone detection approaches are suitable to systems developed by the same teams and for similar purposes. But systems that are developed by different teams but 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.

The next Table 1 shows and compares several tools which were promoted for automatic or semi-automatic approaches.

Table 1 reviews variability analysis tools according to Assunção et al. systematic mapping literature [5]. For each tool showing the analyzing method that can be expert-driven for manual or semi-manual process, static for automatic process and dynamic for analyzing the systems at the running time. Also, we can see if the tool detects information from the input systems like features or information about the variability and the commonality. The Input artifacts can be source code of programming language, requirements which can be the specifications, feature descriptions, customer requests, test suites, documentation, Design models such as class diagrams, state machines, and entity-relationship database models. Finally, the output, there are different types of artifacts were generated, mostly, feature mapped, namely, the features are given and the detecting process locate the relevant code for each feature, next, features discovered, mostly, extract the feature’s elements. The reports artifact, 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 | Analyzing | Detecting | 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 form of a feature model, needs to be done automatically. We still with three tools according to Table 1, first is BUT4Reuse [9], this tool extracts common blocks of code as a feature, this is appropriate to the scenario of one team has been developed all the systems set implementing the clone-and-own method. Also, FMr-T [10] recovers feature models based on cloning. Thirdly, ArborCraft [11] extracted the feature model mainly based on the textual requirements document of the systems. All these three potential tools don’t have any consideration for variability analysis based on recovering systems behaviors that can be applied not only for systems that were developed by the same team but also systems that have been developed by different teams but still behaving similarly.

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

1. Motivation

One of the considerable gaps of the most literature is suggesting variability analysis method neglecting the analyzing behavior variability, mostly they depend on detecting cloned blocks of code, that can be reasonable for systems that developed by the same team using the clone-and-own method, but it is not sufficient for systems that developed by different teams. Whatever at both aspects they can still share similar behaviors. Furthermore, detecting the behaviors is important for discovering features and creating feature models, this is another lack, few literatures proposes detecting features and creating feature model, generally, they suggest feature location [6] for mapping given feature to blocks of code, and few of them offer to create feature model but based on cloned code blocks with unclear or incomprehensible features’ names.

1. Goals

The main goal is to promote a systematic behavior variability analysis method for software systems set either it was developed by the same team or a different team, in which the extracted domain model 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 the feature model will be built according to the extracted domain behavior?

* 1. Methodology

For answering the RQs and achieving the goal, we invent a similarity comparator that can detect functionalities and behaviors from the source code of software systems. Then filtering and verifying which extracted behaviors can be part of the final domain behaviors. Finally, analyze the dependencies of the extracted domain behaviors and build a feature model.

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

1. Suggested method

Our approach consists of four main parts to reach the domain behavior variability analysis as demonstrated in Figure 1. The input is the source code of a systems family, currently java products, (1) first stage, parsing the input artifact, operations descriptors, that describe the operation’s signature (shallow descriptor), and the attributes that were used or were modified (deep descriptor). (2) Then similarity will be calculated among these operations by computing similarity between the shallow part of each couple operation and between the deep descriptor of both, where the results are described by triple Polymorphism-inspired mechanisms (Parametric, Subtyping, and *Overriding*). *Parametric* resemble a high similarity (similar shallow, similar deep), *Subtyping* is moderate similarity (similar shallow, low deep similarity), and finally, *overriding* where just the Shallow descriptor is similar. The output is organized in a graph where the vertices are the product’s artifact, it can be operations as the low-level or higher like classes, packages, or products. The edges are weighted by triple values (p, s, o) describe the similarity in form of (Parametric, subtyping, and Overloading respectively – PSO). Next as a similarity metric, we adopted MCDM (Multi-Criteria Decision Making) that utilized for hierarchical clustering of similar artifacts using the triple similarity mechanisms PSO. The resulting clusters produced candidate domain behaviors. The previous results will be identified and classified in the third (3) step by supervised machine learning model after gathering additional characteristics, Size (no. of: instructions, methods, classes, and projects) and Flow (invoked and invoked by), besides to the PSO for each candidate domain behavior. Finally (4) the Dependency Analysis step organized 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 at [16]. The classification distinguished between three types of domain behaviors, *local* domain behavior that derived just from a single domain element (MCDM-aware cluster), *global* when the domain behavior can be a combination that derived from more than one domain element (MCDM-aware cluster), and *irrelevant* domain behavior, when there is not any domain element that can be derived from.

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 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 are described in Figure 1. For the supervised machine learning stage, we organized a training dataset by classifying behaviors manually for 5 apo-games projects then selecting the best appropriate classifier with related optimum parameters for high accuracy. Then 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 were developed by different student teams in 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

For preparing the approach to automatic execution we selected 5 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 5 games by different researchers for different purposes [16]. The feature model mainly represents concept elements and none of them is behavior. So, 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 Local and Global classes get high precision and recall. The Irrelevant type was not detected because of a very low number of them. So, we decided to deal with irrelevant classified domain behavior as global. Finally, we ran the full approach on the two data set 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-Games dataset, 96 of them were classified as local domain behaviors, whose 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. Besides, 21 domain behaviors were classified as global, which referred to common game behaviors as threads that used for different purposes in the context of games like animations, multi-tasking, parallel playing. In addition, behaviors common to button and mouse interactions, behaviors for properties storage (I/O). The rest, 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 them classified as local domain behaviors which represent the high-level monopoly game elements such as users (admin and players), questions and answers, assets, jail, money, and tile. The rest, 6 global behaviors referred to rounds handling, data storage (I/O), and asset status handling. Three were classified as irrelevant referred to behaviors for locating, answer checking, and system data handling.

Some explanations for the differences between the two-dataset output, we analyze the model part of the monopoly projects (using MVC – Model, View, Control programming pattern) neglecting the view and control parts. Another difference cause can be due to the clear well-defined requirements of a monopoly game, using the same high-level game concept, furthermore, it’s 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 especially using specific abstract game packages e.g. component and entity, that led to many mandatory domain behaviors. Moreover, another explanation that we used for both datasets same machine learning classifier that was built using a subset of the apo-games dataset.

.

* 1. Conclusions

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

The promising achievement is extracting high-level abstraction as a reasonable feature model due to two main extensions: MCDM-aware hierarchical clustering that select 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 developed for the same matter by different teams (Monopoly), second clone-and-own scenario (ApoGames).

Despite the threats of validity that can result from using a sub-set of Apo-Games to train the supervised machine learning model or deficiency of approaches that detect behaviors or systematically create feature models for these datasets. The approach still extracts the main behavior and related elements of the product's domain maybe by different levels as has been got from the previous datasets.

1. Summary

In this thesis, we introduced an automatic holistic behavior variability analysis approach. The approach extracted domain model in form of a feature model from the low-level implementation of software systems family, either developed by the same team or different teams. It is done through 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 by MCDM method to cluster similar operation to be domain behavior candidates, (3) then they were classified into local, global and irrelevant utilizing supervised machine learning, finally, (4) dependency analyzing for classified domain behaviors that are considered as features which were organized by feature model. The approach was implemented by VarMeR[17] tool and evaluated by two datasets from the games domain.

* 1. Contributions

While major of the literature promotes variability analysis based on detecting cloned code blocks that done by the clone-and-own scenario that used generally within the same developing team, our approach introduces variability analysis based on behavior extraction from the low-level source code of software systems set that can be developed by the same team or different teams, then creates domain model in form of the feature model. Here we promote a method where its 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 feature from the design stage or proposed by experts, the remaining mission just to locate the code for each feature, namely mapping process.

The resultant feature model is considered a 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, so there are threats of validity concerning the evaluation, the validity procedure, approach stages, outcomes, selected approach parameters, and using third-party specific algorithms.

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

 Second, the feature model outcome can’t be compared to another approach outcome due to two reasons. First, our model represents “behavioral features” and there are no competing approaches to detecting behaviors. Second, there is no even manual feature model for the two-evaluation datasets. Just for a subset of Apo-games found a very high-level manual feature model that used us for the manual classification while preparing the machine learning training set, but for the monopoly dataset, there is no manual or automatic feature model.

Third, despite our process being automatic we still need to prepare the machine learning stage individually, that’s meant to organize a sufficient training set and select a better classifier and its optimum parameters, therefore we need domain expert involvement for manual classification and to select the optimum classifier.

Fourth, also related to the machine learning stage, there is no trusted professional training dataset specialized in domain behaviors, which can be reliable for training machine learning. Current we embedded classifier optimized on the training set that we prepared using a subset of Apo-games, so it specialized to games domain. We use it also for the Monopoly dataset, maybe 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, optimizing parameters and third-party methods used by the approach’s stages. Our approach used many algorithms and methods in its different stages, for example, semantic similarity method and related threshold, MCDM method, and selected weights, polymorphism-inspired mechanisms threshold. All these parameters need to be optimized systematically.

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

* 1. Future research

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

First, there are several directions to improve the current approach. We need to build a trusted training dataset to classify domain behaviors utilized by the machine learning stage, maybe to support various domains. In addition, support self an automatic process for selecting the best classifier and its optimum specifications. Another possible improvement, the clustering process is by examining other graph-clustering algorithms instead MCDM-SAW method to improve the performance and the expected results. 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 was confirming 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. Not only, but 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, the approach performance was neglected along all the stages, so, we intend to assess the performance for each stage and suggest ways for improvements.

Fourth and finally, to move forward towards SPL (Software Product Line). After the analyzing and detecting phases still perform the transformation stage, to suggest refactoring way based on extracted feature model to produce 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 accompaniment patiently all 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.