A Variability-Driven Analysis Method for Automatic Extraction of Domain Behaviors

Iris Reinhartz-Berger^(✉) and Sameh Abbas

Information Systems Department, University of Haifa, Haifa, Israel
iris@is.haifa.ac.il, samih1079@gmail.com

Abstract. Domain engineering focuses on modeling knowledge in an application domain for supporting systematic reuse in the context of complex and constantly evolving systems. Automatically supporting this task is challenging; most existing methods assume high similarity of variants which limits reuse of the generated domain artifacts, or provide very low-level features rather than actual domain features. As a result, these methods are limited in handling common scenarios such as similarly behaving systems developed by different teams, or merging existing products. To address this gap, we propose a method for extracting domain knowledge in the form of *domain behaviors*, building on a previously developed framework for behavior-based variability analysis among class operations. Machine learning techniques are applied for identifying clusters of operations that can potentially form domain behaviors. The approach is evaluated on a set of open-source video games, named apo-games.

Keywords: Domain engineering · Systematic reuse · Variability analysis

1 Introduction

As systems grow and evolve over time and system development become challenging, it may help learn from existing relevant systems in the domain and create flexible, adaptable, and reusable artifacts. This can be done by performing domain engineering, which focuses on modeling common (and variable) knowledge in an application domain¹ for the purpose of understanding phenomena in the domain and supporting systematic reuse [9]. As this is a tedious and error-prone task, different methods have been proposed to automate it, e.g., by detecting or extracting features – prominent or distinctive user-visible aspects, qualities or characteristics of a system or systems [2].

Most automatic feature extraction methods assume similar system variants that were typically created following cloning scenarios, e.g., clone-and-own in which variants are formed by copying existing artifacts and adapting them to the requirements at hand. This assumption may narrow the uses to various software evolution scenarios, neglecting common scenarios, such as systems developed by different teams yet sharing similar

¹ The term domain has many interpretations: business domain, environment, application domain and more. Hereafter, we refer to domain as application domain.

behaviors or attempts to integrate and merge existing products. Moreover, most of the existing methods provide low-level features rather than actual domain features [6] and hence their ability to help develop and maintain systems, as well as to promote reuse, is limited.

Of the different kinds of features, domain behaviors may help extract reusable services and interfaces. We define *domain behavior* as a transformation of state variables (attributes) of interest in the domain of discourse. Consider as an example the domain of applications for renting physical spaces, such as offices and houses. *Renting* can be considered a domain behavior, which transforms a state variable named *space_status* from *available* to *occupied*. This observation calls for supporting corresponding renting services as the variety of physical spaces and renting policies may be large.

In this paper, we aim to identify domain behaviors for the purpose of increasing reuse across similarly behaving systems, which do not necessarily share similar implementations. To this end, we build on a previously developed framework for *behavior-based variability analysis*, which identifies three mechanisms among behaviorally similar operations: parametric, subtyping, and overloading. Using this framework, we apply machine learning techniques for identifying clusters of operations that form domain behaviors. Particularly, our method gets object-oriented code of different systems and returns a list of domain behaviors that need to be maintained for the management of the input systems and the development of additional systems in the domain. The approach is evaluated using a set of apo-games, which are open-source video games developed in JAVA [14].

The rest of the paper is structured as follows. Section 2 presents the existing behavior-based variability analysis framework, while Sect. 3 introduces the suggested approach for extracting domain behaviors. Section 4 presents preliminary results and Sect. 5 reviews and discusses the related work with respect to the current research. Finally, Sect. 6 summarizes and highlights some future research directions.

2 Behavior-Based Variability Analysis

Features in general and domain behaviors in particular can be extracted from different system artifacts, such as domain information, requirements, design models, and source code [2]. The most available and reliable artifacts are source code, as they can be found in open source or proprietary repositories and represent faithfully the actual system intensions and functionality. Many of the source code artifacts are written in an object-oriented language, whose main elements are classes that exhibit operations.

In a previous work, we have suggested a tool-supported framework for analyzing variability of behaviors in object-oriented code [22]. The tool, named VarMeR – Variability Mechanisms Recommender [21], follows a three-step process: behavior extraction, behavior comparison, and variability analysis. These steps are briefly overviewed and exemplified next.

2.1 Behavior Extraction

Taking an external point of view, a (system) *behavior* is a transformation of state variables (attributes) from an initial state to a final state as a result of an external event². A behavior is represented via two descriptors, shallow and deep, as defined below.

Definition 1. A *shallow descriptor* depicts the interface of the behavior. Formally expressed, a shallow descriptor is a pair (inOut, params), where *inOut* is a pair (bn, rt) – *bn* is the behavior name and *rt* is its returned type, and *params* is a set of pairs (pn, pt) – *pn* is the name of a parameter passed to the behavior and *pt* is its type.

Definition 2. A *deep descriptor* represents the transformation done to the state variables as a result of the behavior. Formally expressed, a deep descriptor is a pair (attUse, attMod), where *attUse/attMod* are sets of pairs (an, at) – *an* is the name of an attribute (state variable) used (read)/modified (written) by the behavior and *at* is its type, respectively.

As an example, assume two behaviors: office renting and house renting³. *Office renting* gets the client who rents the office and updates accordingly the clients list and the office availability status. Availability is checked based on the range of possible number of employees and the office status. *House renting* also gets the customer details, but updates only the customers list. Availability is checked based on the number of available beds. Both behaviors return whether they succeed or fail. Table 1 presents the shallow and deep behaviors of these operations.

Table 1. Examples of behavior representation

Descriptor	Component	Office renting	House renting
Shallow	InOut	(rent, Boolean)	(rent, Boolean)
	Params	(c, Client)	(c, Customer)
Deep	AttUse	(clients, ArrayList) (minEmployees, int) (maxEmployees, int) (status, OfficeStatus)	(customers, ArrayList) (beds, int)
	AttMod	(clients, ArrayList) (status, OfficeStatus)	(customers, ArrayList)

The behavior extraction is done through common static program analysis techniques. Particularly, we use Program Dependence Graph representation [12] to extract the deep descriptors from class operations in object-oriented code.

² The terms states and external events are defined in Bunge-Wand-Weber ontology as follows [28]: a *state* is a vector of values of state variables of a thing at a particular point in time; and *external event* is a change in the state of a thing as a result of an action of another thing.

³ Due to space limitations, code examples from our running example are not included in the paper and can be found at <https://sites.google.com/is.haifa.ac.il/varmer/> (additional).

2.2 Behavior Comparison

In this step, the extracted behaviors are compared using similarity measures. The comparison of the low-level components, namely, *inOut*, *params*, *attUse*, and *attMod*, may be based on semantic nets or statistical techniques [17], whereas behavior comparison is done component-pairwise, namely, behaviors which have similar shallow and deep descriptors are the most similar. We categorize similarity mappings as follows.

Definition 3. Let $B1, B2$ be the components of the shallow/deep descriptors of behaviors $b1, b2$, respectively. Given a similarity mapping $m: B1 \times B2 \rightarrow \{0,1\}$, the corresponding behavior descriptors are claimed to follow⁴:

1. *USE* (abbreviation for use-as-is) – for each $c1 \in B1$ there is exactly one $c2 \in B2$ such that $m(c1, c2) = 1$ and for each $c2 \in B2$ there is exactly one $c1 \in B1$ such that $m(c1, c2) = 1$.
2. *REF* (abbreviation for refinement) – there is at least one $c1 \in B1$ such that there are $c2, c2' \in B2$ satisfying $m(c1, c2) = m(c1, c2') = 1$.
3. *EXT* (abbreviation for extension) – there is at least one $c2 \in B2$ such that there is no $c1 \in B1$ satisfying $m(c1, c2) = 1$.

The above mappings allow for a variety of relations between behaviors. Of those, behaviors whose shallow descriptors are related via *USE* are of special interest, since they intuitively correspond to the case when the behavior intensions as manifested by their interfaces are similar, but not necessarily identical. We thus distinguish between three types of potential mechanisms. These mechanisms, which are inspired by the notion of polymorphism in object-oriented programming, are defined below.

Definition 4. Let $b1, b2$ be two behaviors and m – a similarity mapping on their components. If the behavior shallow descriptors follow *USE*, then the behaviors are claimed to follow:

1. *Parametric* – if the behaviors' deep descriptors also follow *USE*.
2. *Subtyping* – if the behaviors' deep descriptors follow *REF* and/or *EXT*.
3. *Overloading* – if the behaviors' deep descriptors follow neither of *USE*, *REF*, or *EXT*.

Returning to our example, consider the office and house renting behaviors. They can be considered subtyping, as their shallow descriptors are related via *USE*, while their deep descriptors via *EXT* (office status has no counterpart) and *REF* (beds can be mapped to both min and max employees, as all are meant to estimate space capacity).

2.3 Variability Analysis

In the last step, the comparison results are percolated from the behavior level to the entities which exhibit them. In object-oriented terms, classes with similarly behaving

⁴ In practice similarity have values in the range $[0, 1]$. Thus, we define all similarity values exceeding a pre-defined threshold as mapped to 1 ("similar") and all others – to 0 ("different").

operations are similar, and systems with similarly behaving classes are similar. The similarity relations are visualized as graphs whose nodes are entities (systems, classes, or even sub-systems or packages) and the edges represent the potential appropriateness of applying the different polymorphism-inspired mechanisms: parametric, subtyping, and overloading.

Definition 5. A *similarity graph* is a graph $G = (V, E; w)$, where:

- V is a set of nodes of the same type (e.g., systems, classes, or operations). Each node holds its origin (e.g., system.class.operation).
- $E \subseteq V \times V$ is a set of edges.
- $w: E \rightarrow [0,1]^3$ is a weighting function that maps each edge to a triplet representing the parametric, subtyping, and overloading similarities of the connected vertices, respectively.

The weighting function on the level of behaviors (operations) always returns 1 (if the edge exists). On higher levels of abstractions, e.g., classes or systems, the weighting function reflects the fractions of operations which have counterparts. For example, let C_1 and C_2 be two classes exhibiting operations $\{o_{1i}\}_{i=1..n}$, $\{o_{2j}\}_{j=1..m}$, respectively. Then:

$$w(C_1, C_2) = \frac{|\{o_{1i} \in C_1 : \exists o_{2j} \in C_2 s.t. w(o_{1i}, o_{2j}) = 1\}| + |\{o_{2j} \in C_2 : \exists o \in C_1 s.t. w(o, o_{2j}) = 1\}|}{m + n}.$$

An example of VarMeR outcome at the class level is shown in Fig. 1. Each system is visualized in a unique color. All classes depicted in yellow (House, Amenity, and Customer), for example, belong to RentCom. The classes House and Room of FindRoommate, Office of WeWork, and House of RentCom similarly behave, as they all exhibit renting and availability checking operations. The two classes named Amenity (one of WeWork and the other of RentCom) and the class Facility of FindRoommate also share similar behaviors for managing amenities/facilities.

The class Roommate on the other hand is not similar (enough) to the classes Client and Customer, due to behaviors related to gender preferences exhibited only by the former class. This VarMeR outcome implies which types of assets need to be developed and reused in the systems: physical spaces (houses, rooms, and offices), amenities/facilities, and two types of stakeholders (clients/customers and roommates). Note that the slide bars at the top of the screen enables separately controlling similarity thresholds for each one of the mechanisms, directly affecting VarMeR recommendations. The sign “-” denotes a value lower than the corresponding specified threshold. Zooming into the classes, VarMeR can show the corresponding similar behaviors (not shown here, due to space limitation).

Fig. 1. A snapshot of VarMeR

3 The Suggested Approach for Extracting Domain Behaviors

VarMeR results in many recommendations. In order to extract (system) behaviors “of interest” in the domain of discourse rather than just similarly behaving operations, we apply machine learning. An overview of the suggested approach is depicted in Fig. 2. It is composed of three steps: (i) pre-processing – where behavior is extracted after the code is cleaned, e.g., by removing redundant code which may result from applying a cloning scenario; (ii) clustering similar behaviors, where sets of similar operations are suggested as potential domain behaviors; and (iii) domain behavior mining, where meaningful domain behaviors are extracted using machine learning techniques based on metrics of operations’ characteristics.

The first pre-processing step was elaborated above. It further uses static analysis techniques to remove trivial operations, such as getters and setters, and redundant operations which may result from cloning. The rest of the section is devoted to the two other steps – clustering similar behaviors and mining domain behaviors, which together provide the core contributions of the suggested approach.

3.1 Clustering Similar Behaviors

In [23] we introduced an extension of the hierarchical agglomerative clustering algorithm for grouping similar behaviors. As behaviors may be related via different similarity types, namely, parametric, subtyping, and overloading, and hence have different similarity values, we suggested incorporating multi criteria decision making (MCDM) methods [27] into a clustering algorithm in order to group similar behaviors. A typical input of MCDM is a decision matrix, whose rows are the alternatives to be considered and its columns are the criteria. The cell $[i,j]$ holds the evaluation given to alternative i with

Fig. 2. An overview of the suggested approach

respect to criterion j . SAW – Simple Adaptive Weighting, for example, is an intuitive decision-making technique in which calculation is established based on a simple addition of scores that represent the goal achievement under each criterion, multiplied by some predefined weights [19]. In our context, we can weight parametric higher than subtyping and subtyping higher than overloading, in order to align preferences with the less effort required for adaptation. Table 2 exemplifies *part* of a decision matrix for selecting edges from the graph depicted in Fig. 1.

Table 2. *Part* of a decision matrix for selecting edges from the graph in Fig. 1

Alternative	p	s	o	SAW score
(RentCom.Customer, WeWork.Client)	1	0	0	3.00
(RentCom.Amenity, WeWork.Amenity)	1	0	0	3.00
(RentCom.Facility, WeWork.Amenity)	0.5	0	0	1.50
(RentCom.House, FindRoommate.Room)	0.14	0.14	0	1.03
(FindRoommate.Room, WeWork.Office)	0.14	0.14	0	1.03
(WeWork.Office, FindRoommate.House)	0	0.18	0	0.78

As similarity between operations may be accidental, namely, operations that happen to have similar interfaces and handle (use and modify) a similar set of state variables, we suggest here analyzing behavior similarity in the context of classes which own the operations. Listing 1 presents the pseudo code of the suggested clustering method for

identifying similarly behaving classes. This method incorporates MCDM considerations, namely, it continues merging clusters which are suggested by a MCDM technique as best alternatives till a predefined threshold is reached.

Given a cluster of similarly behaving classes, we identify its potential domain behaviors as the maximal induced connected subgraphs of operations.

Inputs: G – a similarity graph at the class level, th – a similarity threshold, lnk – a linkage criterion (e.g., min, max, or avg for single, complete, or average linkage, respectively)

Used structure:

C is a set of clusters, where each cluster is a set of classes (namely, C is a set of sets).

D is a decision matrix

S is a triplet $(C1, C2, score)$, where $C1, C2$ are clusters and score is a number.

Used functions:

$C.initialize(G)$ – creates a set of $|V|$ clusters, in each cluster a single node from the graph G .

$D.create(C, G, lnk)$ – creates a decision matrix based on the given set of clusters, the similarity graph G , and the linkage metric lnk .

$MCDM(D)$ – operates the MCDM methods on the decision matrix D and returns the alternative with the highest score including its score.

$C.merge(C1, C2)$ – returns the set of clusters C , where $C1$ and $C2$ are merged and all other clusters are the same.

$C.initialize(G)$

Repeat

$D.create(C, G, lnk)$

$S = MCDM(D)$

 If $(S.score > th)$

$C.merge(S.C1, S.C2)$

Until $(S.score \leq th)$

return C

Listing 1. Pseudo code of clustering and identifying similarly behaving classes

Definition 6. Given a similarity graph $G = (V, E; w)$ and a cluster of similarly behaving classes $C \subseteq V$. A *candidate domain behavior* for cluster C is a maximal connected subgraph of $G|_C$ (i.e., the reduction of the similarity graph G to vertices in C).

Figure 3 exemplifies candidate domain behaviors for space rental applications.

Fig. 3. Examples of candidate domain behaviors

3.2 Mining Domain Behaviors

The next stage is classifying whether the identified clusters, namely, the candidate domain behaviors, are indeed domain behaviors. The classification is commonly done by a human domain expert, but we explore in this study whether characteristics related to similarity, flow, and size of behaviors may provide good prediction. Thus, we identify three possibilities in which clusters may be mapped to domain behavior(s): specific, general, or irrelevant, as formalized below. For a cluster C and a domain behavior d , we denote $C \rightarrow d$ if C corresponds to d (namely, if a candidate domain behavior is indeed a domain behavior).

Definition 7. Let C be a cluster of similar behaviors (which may be obtained as a result of the previous step).

- C is *specific* if $C \rightarrow d$ for some (single) domain behavior d .
- C is *general* if there are different domain behaviors $d_1, \dots, d_k$ such that $C = C_1 \cup \dots \cup C_k$ and $C_i \rightarrow d_i$ for $i = 1$ to k .
- C is *irrelevant* if C is not specific neither general, i.e., there is no domain behavior d such that $C \rightarrow d$ or $C_i \rightarrow d_i$ where $C_i \subseteq C$.

As the set of domain behaviors is not known while conducting domain engineering, we now aim to build an automated classifier based on the above definition. The features which we hypothesize to be relevant for the classification belong to three categories:

1. Similarity-related: the normalized numbers of parametric, subtyping, and overload relations *within* the cluster.
2. Flow-related: the minimal, maximal, and average numbers of operations invoking and being invoked by operations *within* the cluster.

3. Size-related: the numbers of involved instructions, operations, classes, and systems in the cluster.

We applied a supervised machine learning algorithm – random forest [5], in order to identify whether the above features predict domain behavior classification and to what extent. Some preliminary results are presented and discussed next.

4 Preliminary Results

For a preliminary evaluation of the approach, we used a set of apo-games, which are open-source video games developed in JAVA by an experienced industrial developer over the years 2006–2012. These games are different in their purpose and their code underwent some evolutionary changes over the years. Yet they have a common infrastructure and behavior and hence 20 games have recently been proposed as a variability reverse engineering challenge [14].

We selected two categories of the apo-games: 5 navigation games and 8 puzzle games. The characteristics of these games are summarized in Table 3. For each category, we extracted and created a similarity graph, using the semantic similarity metric proposed in [17]. We further clustered similar behaviors as described in Sect. 3.1, and collected and recorded the values of the features discussed in Sect. 3.2. 60 clusters were generated for the puzzle games and 62 – for the navigation games. Two human experts independently classified the 122 clusters into specific, general, and irrelevant domain behaviors, and discussed their classification till full agreement was reached. See Table 4 for characteristics of the resultant classification.

Overall, the initial candidate domain behaviors generated by our approach seem to be relevant (specific or general). Only few clusters (11.6% of puzzle and 4.8% of navigation) were found irrelevant. These were clusters containing low-level operations which seem unsuitable for being considered as domain behaviors. Several clusters (15% for puzzle and 16% for navigation) were found general, i.e., including operations which span across multiple domain behaviors.

While it may be possible to reduce the numbers of irrelevant and general clusters, we claim that complete avoidance is impossible, as operations and domain behaviors are assumed to be at different levels of abstraction. Thus, for automatization of domain behavior classification, we ran random forest in the WEKA framework [11] with the commonly used 5-fold cross-validation. The results are depicted in Table 5.

High F-measure values were obtained for specific clusters in both domains (0.891 and 0.911 for puzzle and navigation, respectively), demonstrating the potential for correctly extracting domain behaviors automatically. Irrelevant clusters were precisely classified, but had lower recall in both domains (0.286 and 0.667 for puzzle and navigation, respectively). Both precision and recall for the general category in both domains were relatively low. These results may be attributed to the low numbers of examples of these categories in the sample.

Table 3. Characteristics of the apo-games selected for evaluation (The values at the SLOC column are presented after cleaning redundant code, which is not used in given games and resulted from cloning the code of similar games.)

	Name	SLOC	Goal
Puzzle	ApoBot	5,029	To program a bot to turn the blue fields into red
	ApoNotSoSimple	5,770	A clever but tricky puzzler with a possibility to build levels and publish them
	ApoRelax	4,934	To rearrange parts so that they give a picture
	ApoSimple	17,865	To eliminate colored circles
	ApoSlitherLink	4,986	To draw a closed path without intersections and ramifications of edges between the points of a grid
	ApoSnake	3,765	A snake game with more than 30 levels and a possibility to upload more
	ApoStarz	5,039	To return stars to their proper places
	ApoSudoku	4,119	A Sudoku game
Navigation	ApoCommando	8,053	To make the player reach a goal by entering commands (such as “jump”) in a command line
	ApoIcarus	4,466	To reach the sun with Icarus by jumping from cloud to cloud without falling down or getting caught by an opponent
	ApoIcejumpReloaded	6,624	Not to fall into the water as long as possible
	ApoImp	3,804	To reach a Christmas tree, move gifts, beam around and climb
	ApoMario	15,801	To navigate to the end of the level

Table 4. Numbers of the candidate domain behaviors

	Puzzle games	Navigation games
Specific	44	49
General	9	10
Irrelevant	7	3
Total	60	62

The most meaningful features for classification are mainly size-related (particularly the number of involved classes within the clusters) and flow-related (particularly the number of operations invoking operations within the clusters). These features look into the impact of the clustered operations across systems (apo-games) and thus are good predictors for domain behaviors. However, it should be noted that both of our considered

Table 5. Classification results

	Class (Category)	Precision	Recall	F-Measure	ROC area	Strongest features
Puzzle	Specific	0.854	0.932	0.891	0.807	<ul style="list-style-type: none">• size-related (# of classes, systems)• similarity-related (overloading)• flow-related (invoked by)
	General	0.600	0.667	0.632	0.855	
	Irrelevant	1	0.286	0.444	0.642	
	Weighted avg.	0.833	0.817	0.800	0.795	
Navigation	Specific	0.885	0.939	0.911	0.841	<ul style="list-style-type: none">• size-related (# of classes)• flow-related (invoked by, invokes)
	General	0.625	0.500	0.556	0.862	
	Irrelevant	1	0.667	0.800	0.989	
	Weighted avg.	0.848	0.855	0.848	0.852	

domains are in the context of games. Indeed, we found behaviors common to the two categories of games, including behaviors dealing with buttons, images, mouse, audio/sound and more. Puzzle game behaviors further include level management and string manipulations, whereas navigation game behaviors also address entities, such as obstacles and enemies, locations and motions. Further investigation of generalizability of our method to other domains is needed. Particularly, running experiments in additional domains will help better isolate the important features that are not domain-specific.

5 Related Work

5.1 Variability Analysis

An important context in which domain engineering is commonly conducted is that of software product lines, which are families of systems that share common assets allowing systematic reuse [4, 18]. Apel et al. [1] provide a comprehensive overview of technical terms and techniques in the software product line engineering field from a feature-oriented perspective. Commonly, feature is a basic unit for commonality and variability analysis. In a recent literature review [2], feature detection is identified as the first phase of product line reengineering (followed by variability analysis and artifact transformation). In [13], three related tasks are mentioned: (1) feature identification – for determining the features that exist in a system based on a set of artifacts, revealing their dependencies and documenting the result (e.g., [15]); (2) feature location – for finding the artifacts (commonly source code) that implement a feature [8, 25]; and (3) feature mapping – for documenting the connection between a feature and its implementation (or realization). Of those, our suggested approach is mostly similar to feature identification. However, differently from existing methods, such as But4Reuse [15], which suggests features that are blocks of source code and thus actually low-level features, our approach provides

more high-level domain features in the form of domain behaviors that may better support the development of reusable assets across systems.

Another context for application of variability analysis techniques is clone detection. Clones are segments of code that are similar according to some definition of similarity. Bellon et al. [3] identify three types of clones: Type 1 – exact copy without modifications (except for white space and comments); Type 2 – syntactically identical copy (only variable, type, or function identifiers were changed); and Type 3 – copy with further modifications (statements were changed, added, or removed). Other types of clones, such as semantic clones, structural clones, function clones, and model-based clones, are also mentioned in the literature [20, 24]. The main drawback of such approaches in our context is that they do the comparison (i.e., the similarity analysis) on the low level of implementations. Our approach, which takes a behavioral view, can better deal with analysis of artifacts which similarly behave but are differently designed (e.g., offices and houses).

5.2 Machine Learning in the Context of Systematic Reuse

While an increasing number of works apply machine learning techniques in software engineering [16, 29], works in the context of software reuse are scarcer. In the context of variability analysis, Ghofrani et al. [10] present a conceptual framework for clone detection using convolutional neural networks. In the context of software product lines, Safdar et al. [26] present an approach for mining rules for automatic configuration which combines multi-objective search with machine learning. Stefano and Menzies [7] performed a case study on a reuse data set using three different styles of learners: association rule, decision tree induction, and treatment and found some procedures which should significantly improve the odds of a reuse program succeeding. To the best of our knowledge, our approach is the first to apply machine learning techniques in the context of automatic support of domain engineering.

6 Summary and Future Research

Automatically supporting domain engineering is a challenging problem due to the required expert knowledge which heavily varies from one domain to another. So far most existing methods either assume high similarity of variants, or provide very low-level features which can mainly be used in cloning scenarios. This paper takes a new angle into the problem, proposing a method for extracting domain knowledge in the form of domain behaviors. The method builds on a previously developed framework for behavior-based variability analysis, applying on top of it clustering and machine learning techniques for domain behavior mining. The approach is evaluated on a subset of apogames, open-source video games recently proposed as a variability reverse engineering challenge. Overall the results are promising in the sense of obtaining automatic classifiers for meaningful domain behaviors. Yet the obtained results are highly dependent on the availability of datasets annotated by human experts and the correctness of their annotations. One immediate direction for future research is considering unsupervised learning approaches and convolutional neural networks to reduce this dependency.

Another immediate direction for future research relates to improvement of the method design and its evaluation, including: (1) exploring additional clustering, MCDM, and machine learning techniques; (2) performing evaluation in additional domains, e.g., in the automotive or manufacturing sector; (3) conducting more large scale empirical studies for evaluating the suggested approach in different settings; and (4) comparing the approach to the state-of-the-art feature extraction methods.

Finally, we intend to further extend the approach to support systematic reuse, e.g., by guiding the creation of reusable artifacts or services, using the identified domain behaviors. We envision this to be done using code manipulation techniques such as refactoring. Extension of the behavior-based framework is also considered, in order to support further mechanisms whose shallow descriptors are not related via USE.

Acknowledgement. The authors would like to thank Anna Zamansky for her advice to the formal parts and her help in the evaluation of the approach.

References

1. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer, Heidelberg (2016). <https://doi.org/10.1007/978-3-642-37521-7>
2. Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A.: Reengineering legacy applications into software product lines: a systematic mapping. *Empir. Softw. Eng.* **22**(6), 2972–3016 (2017). <https://doi.org/10.1007/s10664-017-9499-z>
3. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* **33**(9), 577–591 (2007)
4. Clements, P., Northrop, L.: Software Product Lines. Addison-Wesley, Boston (2002)
5. Cutler, A., Cutler, D.R., Stevens, J.R.: Random forests. In: Zhang, C., Ma, Y. (eds.) *Ensemble Machine Learning*, pp. 157–175. Springer, Boston (2012). https://doi.org/10.1007/978-1-4419-9326-7_5
6. Debbiche, J., Lignell, O., Krüger, J., Berger, T.: Migrating the Java-based apo-games into a composition-based software product line. In: 23rd International Systems and Software Product Line Conference (SPLC), Challenge Track (2019)
7. Di Stefano, J.S., Menzies, T.: Machine learning for software engineering: case studies in software reuse. In: 14th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2002, Proceedings, pp. 246–251. IEEE, November 2002
8. Dit, B., Reville, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. *J. Softw. Evol. Process* **25**(1), 53–95 (2013)
9. Falbo, R.D.A., Guizzardi, G., Duarte, K.C.: An ontological approach to domain engineering. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, pp. 351–358. ACM, July 2002
10. Ghofrani, J., Mohseni, M., Bozorgmehr, A.: A conceptual framework for clone detection using machine learning. In: IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEL), pp. 0810–0817. IEEE, December 2017
11. Holmes, G., Donkin, A., Witten, I.H.: Weka: a machine learning workbench (1994)
12. Krinke, J.: Identifying similar code with program dependence graphs. In: Proceedings Eighth Working Conference on Reverse Engineering, pp. 301–309. IEEE, October 2001
13. Krüger, J., Berger, T., Leich, T.: Features and how to find them: a survey of manual feature location. In: Mistrik, I., Galster, M., Maxim, B. (eds.) *Software Engineering for Variability Intensive Systems: Foundations and Applications*. Taylor & Francis Group, LLC/CRC Press (2018)

14. Krüger, J., Fenske, W., Thüm, T., Aporius, D., Saake, G., Leich, T.: Apo-games: a case study for reverse engineering variability from cloned Java variants. In: Proceedings of the 22nd International Conference on Systems and Software Product Line-Volume 1, pp. 251–256. ACM, September 2018
15. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Bottom-up technologies for reuse: automated extractive adoption of software product lines. In: Proceedings of the 39th International Conference on Software Engineering Companion, pp. 67–70. IEEE Press, May 2017
16. Menzies, T.: Practical machine learning for software engineering and knowledge engineering. In: Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals, pp. 837–862 (2001)
17. Mihalcea, R., Corley, C., Strapparava, C.: Corpus-based and knowledge-based measures of text semantic similarity. American Association for Artificial Intelligence, AAAI 2006, pp. 775–780 (2006)
18. Pohl, K., Böckle, G., van Der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005). <https://doi.org/10.1007/3-540-28901-1>
19. Qin, X., Huang, G., Chakma, A., Nie, X., Lin, Q.: A MCDM-based expert system for climate-change impact assessment and adaptation planning – a case study for the Georgia Basin. Can. Expert Syst. Appl. **34**(3), 2164–2179 (2008)
20. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: a systematic review. Inf. Softw. Technol. **55**(7), 1165–1199 (2013)
21. Reinhartz-Berger, I., Zamansky, A.: VarMeR – a variability mechanisms recommender for software artifacts. In: CAiSE-Forum-DC, pp. 57–64 (2017)
22. Reinhartz-Berger, I., Zamansky, A.: A behavior-based framework for assessing product lineability. In: Krogstie, J., Reijers, Hajo A. (eds.) CAiSE 2018. LNCS, vol. 10816, pp. 571–586. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91563-0_35
23. Reinhartz-Berger, I., Abbas, S., Zamansky, A.: Towards polymorphism-inspired recommendation on software product line artifacts. In: Proceedings of Models and Evolution Workshops, MODELS 2019 (2019)
24. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (2009)
25. Rubin, J., Chechik, M.: A survey of feature location techniques. In: Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., Bettin, J. (eds.) Domain Engineering, pp. 29–58. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36654-3_2
26. Safdar, S.A., Lu, H., Yue, T., Ali, S.: Mining cross product line rules with multi-objective search and machine learning. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1319–1326. ACM, July 2017
27. Velasquez, M., Hester, P.T.: An analysis of multi-criteria decision making methods. Int. J. Oper. Res. **10**(2), 56–66 (2013)
28. Wand, Y., Weber, R.: On the deep structure of information systems. Inf. Syst. J. **5**(3), 203–223 (1995)
29. Zhang, D., Tsai, J.J. (eds.): Machine Learning Applications in Software Engineering, vol. 16. World Scientific Inc. Publishers, USA (2005)