Information Retrieval Based Feature Analysis for Product Line Adoption in 4GL Systems

András Kicsi, László Vidács and Árpád Beszédes
Department of Software Engineering and
MTA-SZTE Research Group on Artificial Intelligence
University of Szeged
Szeged, Hungary
{akicsi, lac, beszedes}@inf.u-szeged.hu

Ferenc Kocsis and István Kovács
SZEGED Software Ltd.
Szeged, Hungary
{kocsis.ferenc, kovacs.istvan}@szegedsw.hu

***Abstract*—New customers often require custom features of a successfully marketed product. As the number of variants grow, new challenges arise in the maintenance and evolution activities. Software product line (SPL) architecture is a timely answer to these challenges. The SPL adoption however is a large one time investment that affects both technical and organizational issues. From the program code point of view, the extractive approach is appropriate when there are already several product variants. Analyzing the feature structure, the differences and commonalities of the variants lead to the new common architecture. In this work in progress paper we report initial experiments of feature extraction from a set of product variants written in the Magic fourth generation language (4GL). Since existing approaches are mostly designed for mainstream languages, we adapted and reused reverse engineering approaches to the 4GL environment. We followed a semi-automatic feature extraction method, where the higher level features are provided by domain experts. These features are then linked to the internal structure of Magic applications using a textual similarity (IR-based) method. We demonstrate the feasibility of 4GL feature extraction method and validate it on two variants of a real life logistical system each consisting of more than 2000 Magic programs.**

***Keywords*-Product lines, SPL, feature extraction, Magic, 4GL, information retrieval, LSI**

I. Introduction

When the number of product variants increases, a natural step towards more effective development is the adoption of product line architecture. This holds for systems developed in fourth generation languages (4GLs) as well. However this very high level paradigm needs special treatment. In the traditional sense there is no source code, rather the developer sets up user interface and data processing units in a development environment. The flow of the program follows a well-defined structure. 4GL environments today play an important role in maintaining crucial, mission critical legacy software. In addition, evolving languages offer new (web based) technologies and the efficiency of a real RADD (Rapid Application Development and Deployment) environment. These environments usually offer ready solutions for problems in developing a typical business application (e.g. connecting to database, supporting different database management or operating systems, managing data, etc.). The Magic XPA language [1] has come a long way since its release decades ago. Migrating to new technologies was the key factor in its evolution.

Magic systems however face other challenges than pure technological questions. Maintaining and releasing similar new products accumulates significant overhead over time. Product line architecture offers a timely solution for these challenges [2]. Product line adoption is usually approached from three directions: the proactive approach starts with domain analysis and applies variability management from scratch. The reactive approach incrementally replies to the new customer needs when they arise.

Finally, the extractive approach analyzes existing products to obtain feature models and build the product line architecture [3]. An advantage of the extractive approach in general is that several reverse engineering methods exist to support feature extraction and analysis [4]. Static analysis methods for obtaining structural information and dependencies and the analysis of dynamic execution traces foster feature detection and location activities [5], [6].

In the case of well established Magic systems, where usually there are already a number of systems in production, the extractive approach seems to be the most feasible choice. During the extractive approach the adoption process benefits from systematic reuse of existing design and architectural knowledge. On the other hand, in case of 4GL the reverse engineering tool support is not as advanced as in the case of mainstream, object oriented languages.

The scope of our work is the feature identification and analysis phase, which is a well studied topic in the literature of mainstream languages [5]. Our work is motivated by a research project where product line architecture is to be built based on existing set of products. Our subject is a high market value logistical wholesale system, which is adapted to various domains in the past using clone-and-own method. Although there is reverse engineering support for usual maintenance activities [7], [8], the special structure of Magic programs makes it necessary to experiment with targeted solutions for coping with features. The current paper is a work in progress report of the feature extraction phase. We introduce

an information retrieval (IR) based approach to overcome 4GL specifics, and especially the fact that there are products written in Magic's different main versions of more than 20 years.

The paper is organized as follows. We present the background of our research in the next section by depicting the variability in systems developed by a software company using Magic language. We introduce an information retrieval based feature analysis method for Magic systems, present experiment design and results in Section III. Related work is briefly introduced in Section V, and we conclude our paper in the last section.

II. Background

Adopting software product line architecture is usually a sign of a successful software company. This is a necessary step in the software evolution of a large scale system with a high number of derived specific products. Started more than 30 years ago, the subject system of our analysis has become a leading pharmaceutical logistical wholesale system. During this period of time, more than 20 derived products were introduced at various complexity and maturity levels. Currently these products have independent life cycles and their maintenance is isolated.

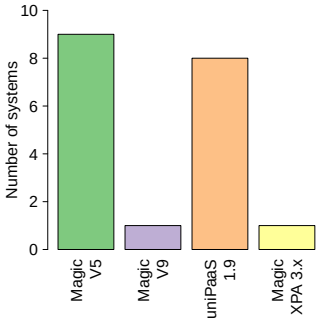

Fig. 1. Total 19 currently active product variants implemented in various versions of Magic

The 4GL environment used to implement the systems requires different approaches and analysis tools than today's mainstream languages like Java [7], [9]. For example there is no source code in its traditional sense. The developers work in the development environment by customizing several properties of programs. Our industrial partner is the developer of market leading solutions in the region, which are built upon a common root and implemented in the Magic XPA 4GL. Over time, both the Magic language and the development environment improved a lot. The language had several main releases, changed its console-based outlook to a modern interface and the underlying architecture is changed to the .NET framework.

The aim of the current project is to form a Magic product line architecture in a semi-automatic way. In the center of the work there are 19 concrete product variants, which are analyzed manually and using experimental analysis tools. Magic analysis tool support is not comparable to mainstream languages, hence this is a research-intensive project. The overall size of the common codebase of product variants is shown in Table I. The first column states that there are 19 currently active variants of the application, while the remaining columns contain the main specifications of the largest variant. Magic is a data-intensive language, which clearly reflects on these values as well, containing a large amount of data tables.

TABLE I
Overview of the common program base of application variants

Variants	Largest application size		
	Programs	Models	Tables
19	4 251	822	1 065

Product variants themselves are written in 4 different language versions as shown in Figure 1. There is a huge difference in the success of various versions. This is also reflected in the figure. The oldest version (Magic V5) is still a stable version, but outdated from many points of view. In case of Magic V5 systems, there is a high demand on the migration to a newer version. A new era is represented by uniPaaS 1.9 systems, where the .NET engine is already used. Most systems are implemented in that version. The newest Magic XPA 3.x line of the language lies closer to the uniPaaS v1.9 systems, hence the latter are in transition to the newest language.

The existing set of products provide appropriate environment for an extractive SPL adoption approach. Characterizing features is usually a manual or semi-automated task, where domain experts, product owners and developers co-operate. Our aim is to help this process by automatic analysis of the relation of higher level features and map program level entities to features.

III. Feature extraction and analysis

Product line adoption first copes with features. In the feature detection phase various artifacts are obtained to identify features in the program. This phase is also called feature location. The analysis phase targets common and variable properties of features and prepares the reengineering phase. This last phase migrates the subject system to the product line architecture.

In this phase of the research project we address the feature location/extraction phase. Our inputs are the high level features of the system and the program code. We apply a semi-automated process as in [4]. High level features are collected by domain experts from the developer company. The concrete task is to establish a link between features and main constituents of the Magic applications. Information retrieval is successfully applied in traceability scenarios for object oriented languages [10]. For our purpose it is appropriate for two main reasons. One of the main challenges of the project is to cope with the substantially different language versions in a uniform way. There are 9 selected products which are the main target of our work. Four of them are v5 systems and five of them are v1.9 systems. Reverse engineering 4GL systems in not well studied topic in literature. Although there exists a common analysis infrastructure for reverse engineering both languages [11], [7], [8], the concrete program models differ.

Applying IR-based solutions make it easier to cope with a 4GL language because they are based on textual features and not on the syntax of the language, and this provides a good way to handle text with less dependence on the code. A good IR-based technique for our specific problem could be Latent Semantic Indexing (LSI), which is a well known and widely used method throughout software engineering.

A comprehensive overview of Natural Language Processing (NLP) techniques – including LSI – is provided by Falessi et al. [12]. In their work several parameters are introduced that may affect the result of an NLP technique. The motivation is to choose the best technique for the product line based analysis of the requirements of a newly developed system of systems. The paper concludes that complex techniques like LSI perform better in case of traceability link recovery scenario, where the query and the corpus belong to different abstraction levels. Since we face this type of problem, we applied LSI to obtain similarity information between features and Magic programs.

A. The Structure of a Magic Application

Magic applications are constructed in the development environment, not relying heavily on coding tasks, therefore there is no traditional source code as a product of the work. This means that our information retrieval methods can work with the names of the various elements of an application. This usually produces quite a small amount of text to work with.

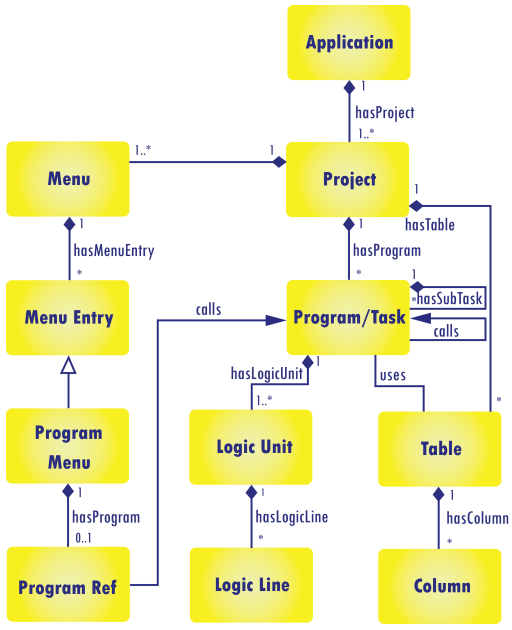

Fig. 2. The most important elements of a Magic application

A Magic application can be most easily described as a tree-structure. Figure 2 shows the most important elements of a typical Magic application with a little more in-depth glance at the Menu aspects we used through our work. A Magic application always consists of one or more Projects, which usually have their own Data Tables accommodating the relevant data of each Project. The most important building blocks of Projects are called Tasks. Tasks could be viewed as the methods of a traditional programming language. The most important distinction here is that each Task can have several subtasks, which are Tasks in their own right. Therefore each task has it's own tree-structure of subtasks. The topmost level of Tasks branching from Projects are called Programs. In our research we mainly worked on this Program level. Aside from subtasks, Tasks usually consist of Logic Units, which represent a series of Logic Lines, providing the base functionality needed, calling Tasks or handling variables.

To utilize a Task, we have to call it. This can occur by the workings of other Tasks or a Menu. The functional distinction of Tasks can be mainly reached through Menus. A Menu is a Project-level element of the application, consisting of several kinds of Menu Entries, of which we would like to distinguish the Program Menus. A Program Menu's purpose is to call a Task when needed, specifically to call the Program stored by it's Task Reference.

IV. Experiments

During our experiments we used the LSI [13] technique widely applied throughout software engineering for various information retrieval tasks. For our purposes, we used the Gensim [14] implementation of the technique. The LSI works with a parameter representing the size of the eigenvector used during its process, during our experiments this was set to the value of 600. LSI is a topic modeling algorithm. This basically means that it can discover clusters of data based on the semantic structures of a body of text. LSI works with a corpus built from documents and with queries containing the data we are curious about. A semantic space is constructed from the corpus with each document occupying a specific point in this space. This point is determined by the semantic structure of each document and the documents with more similar semantic structure occupy points closer to each other. LSI determines the cosine distance of queries from each of these documents, and thus the similarity to each.

In this section we overview the experiments we did as part of this research. We had several Magic application variants on our disposal. We also had a list of the menu-tree used in all of these variants with multiple levels of menu elements. The applications did not use all the menu elements described on the list, merely a subset of it, each using a different set. This also indicates the difference in their functionalities. We also have been provided with a multi-level feature list describing the basic functionalities provided by these applications.

Our main goal was to find the implementing programs of the application for each feature in the list. In most cases, the description of a feature consists of two or three words of Hungarian language text. This provides a rather small text base for each feature, and this usually reflects badly on the result of the LSI similarity, since under these circumstances one mere mention of a word from the feature results in a disproportionally large similarity value, making the LSI over-sensitive to these words. Sadly this over-sensitivity cannot be totally eliminated since it stems from the base behavior of

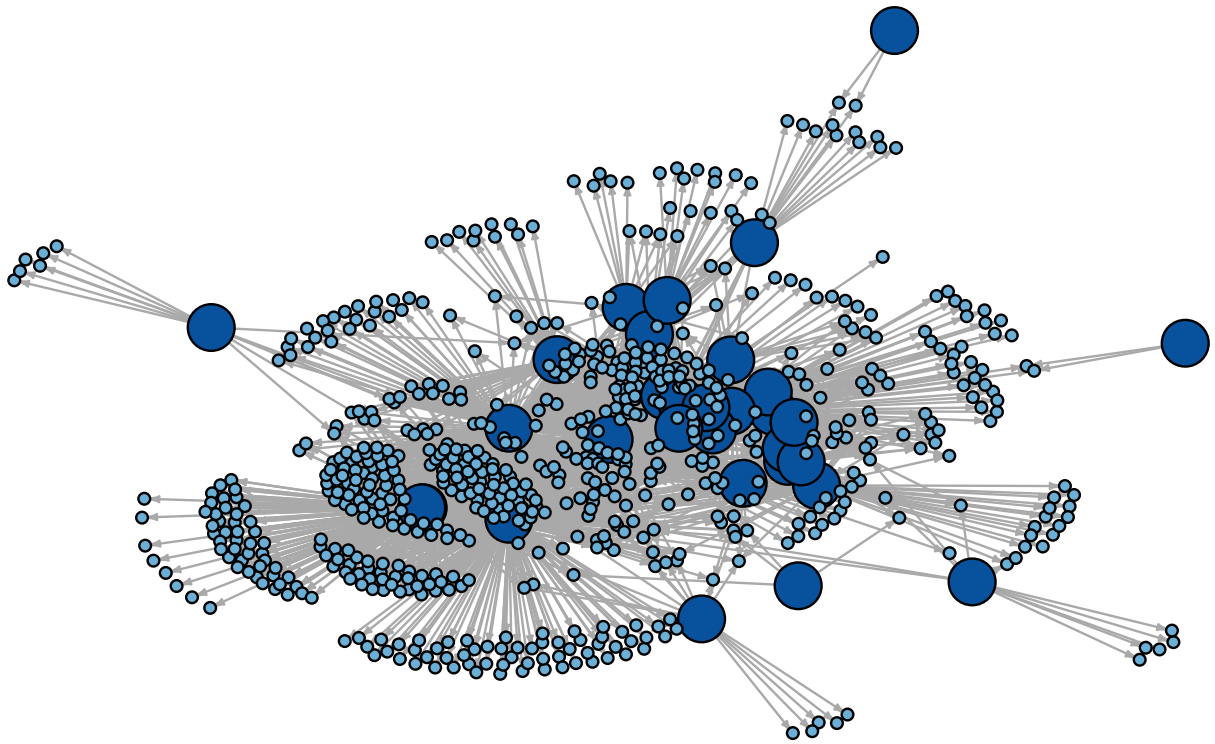

Fig. 3. Extracted features from Magic program code – features (large) and linked programs (small)

the technique. To reduce it, we utilized the other levels of the feature-tree, so that each feature is represented not only by their own textual value, but the value of its parent features and child features too. This provides more versatility to each feature, and still remains fairly loyal to the true meaning of each feature text (since the parent and child features are representing mostly the same functionality the feature does).

The core of our process was the following: We assembled the text of each program using the names of the programs and the elements they contain. This text, and the feature's text alike went through a preprocessing method, in which we used the *magyarlanc* tool [15] for Hungarian lemmatization purposes and we filtered out accentuated letters common in Hungarian text and the different punctuation marks. With the preprocessed text of the programs we built the corpus, and the feature names became the queries for the LSI. The LSI's results were filtered by their determined similarity.

1) Experiment 1: The aim of our first experiment was to validate the feasibility of our process, and provide a success rate of our results. As already mentioned, we had a menu-tree, a subset of which corresponds to the menus used by each application. Each feature invokes one or usually more programs through these menus, covering possibly a rather large base of all the programs of the application variant. This is achieved through the menu elements which are responsible for calling their associated programs. We have also been provided with a tree of each feature and their invoked menus. Through these connections we were able to produce quantifiable results, which can demonstrate the feasibility of the process.

Bug localization is one of the most studied scenarios, which is an appropriate baseline of our experiment. We compare our results to a recent result bug localization result from Thomas et al. [16]. They evaluated the set of top 20 recommended items by LSI and reached 40-70% results. Our results are shown in Table II. The applications of the table represent different product variants of the Magic application. Our results in the table represent the rate of features which had at least one genuinely connecting program assigned to them by our method. These results correspond to the state-of-the-art IR-based bug localization scenarios. Thus, we consider that IR-based approach is feasible on Magic applications with a comparable success as in case of similar scenarios of mainstream languages.

TABLE II
RESULTS OF FIRST EXPERIMENT

Application	Features Present	Features Found	Success Rate
Variant 1	22	11	50%
Variant 2	25	16	64%

2) Experiment 2: The aim of the second experiment was to extract features from the program code by linking high level features provided by domain experts with a set of Magic programs. In practice there can be features which are handled by only one program, and there are also some which are handled by many more than 20 programs, at the top level of features this number could be in the hundreds. Luckily the LSI method can provide a similarity value for each feature-

program pair, so we do not have to resolve on attaining the 20 most similar to each, we can use more intelligent methods, like setting up a similarity threshold.

TABLE III
STATISTICS OF FEATURE-PROGRAM LINKS AT THRESHOLD VALUE 0.45

Application	All Programs	Programs Assigned	Programs Assigned to 1 feat.	Programs Assigned to more	Max Ass.
Variant 1	2719	708 (26%)	208	500	18
Variant 2	2001	377 (19%)	185	192	17

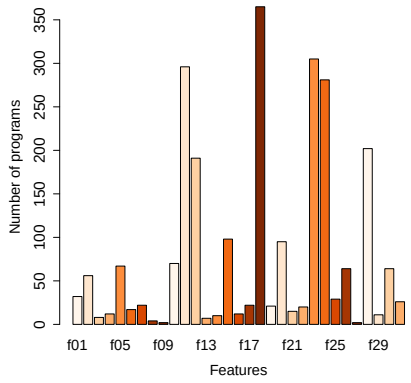

Fig. 4. Distribution of feature sizes (number of linked programs)

The threshold can be a number between 0 and 1; 0 meaning that we allow connections between very dissimilar programs and features, while 1 meaning that we accept only exact similarities, filtering out most of the possible connections, resulting in a very small amount of matches.

The results of this second experiment can be seen in Table III. These results were produced with the similarity threshold set to the value of 0.45. These results of course only represent the statistical side of our output, validation of the connections themselves is still up for manual evaluation done by domain experts, which we are planning for in the future. In Figure 3 we can see the connections our process found in Experiment 2 at the Variant 1 of the application at a similarity level of 0.45. The large circles represent the high level features, while the small circles represent the programs of Variant 1. The arrows represent the connections our process found, namely a large enough textual similarity between features and programs. It can be clearly seen that some features share the same programs, and that each feature is deemed most similar to a different number and set of programs, providing a rather lifelike image. It is also clear that some programs are only connected to one feature, as they presumably only exist to serve one concrete purpose, while others may contribute to more features. The distribution of programs for each feature can be seen in Figure 4.

We have built an experimentational tool-chain to conduct a whole line of experiments with different IR settings (including corpus extraction and similarity thresholds and other parameters). It features a visually pleasing graphical interface, making it easier to locate features in Magic applications. The user interface of this tool can be seen in Figure 5 and Figure 6. Figure 5 displays the tool in an initial state providing the opportunity to set the most basic settings of the process, while Figure 6 shows the result of an experiment, displaying the number of most similar programs for each feature, and optionally listing these programs for the user.

Fig. 5. Configuration window for Magic feature location experiments

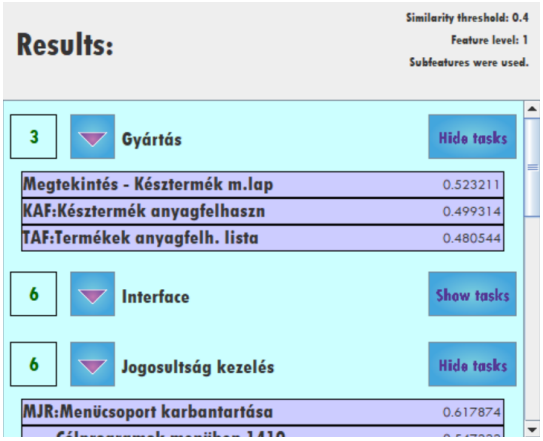

Fig. 6. IR-based feature–program links in a concrete application

V. RELATED WORK

The literature of reverse engineering 4GL languages is not extensive. By the time the 4GL paradigm arisen, most papers coped with the role of those languages in software development, including discussions demonstrating their viability. The paradigm is still successful, on the other hand only a few works are published about the automatic analysis and modeling 4GL or concretely Magic applications. The maintenance of Magic applications is supported by cost estimation and quality analysis methods [17], [18], [11]. Architectural analysis, reverse engineering and optimization are visible topics in the Magic community [9], [19], [8], [7], as well as, after some years of Magic development, migration to object-oriented languages [20].

SPL literature is widespread and there is an observable increasing tendency during the last 8-10 years. All three phases of feature analysis (identification, analysis, and transformation) are tackled by researchers. A recommended mapping study on recent literature on feature location can be read in [5].

Feature models are first class artifacts in variability modeling. Haslinger et al. [21] present an algorithm that reverse engineers a FM for a given SPL from feature sets which describe the characteristics each product variant provides. She et al. [22] analyze Linux kernel (which is a standard subject in variability analysis) configurations to obtain feature models. LSI is applied for recovering traceability links between various software artifacts. The work of Marcus and Maletic [10] is an early paper on applying LSI for this purpose. Eyal-Salman et al. [6] use LSI for recovering traceability link between features and source code with about 80% success rate, but experiments are done only for a small set of features of a simple java program. IR-based solution for feature extraction is combined with structural information in the work of Almsie'deen et al. [23]. This is a promising direction and in case of Magic applications call dependencies are also planned to be used for detailed feature analysis.

Several existing approaches can be adapted to 4GL environment, as we showed in this paper, however none of the above cited papers cope with 4GL product lines directly.

VI. Conclusions

In this paper we reported initial experiments in feature extraction of a Magic 4GL application. The work is motivated by the several product variants that are currently maintained in parallel. Since there are several products written in different language versions, we conducted experiments with an information retrieval method. To validate the method for Magic applications, the links between menus and programs were recovered with a successful rate acceptable in the state-of-the-art of bug localization for Java programs. The method is then applied to extract features in Magic by establishing links between high level features and application programs. The preliminary results are promising, however manual validation by domain experts is still needed. The next step towards the Magic product line architecture is to analyze variability of the above identified features across existing applications.

Acknowledgment

This research was partially supported by the Hungarian national grant GINOP-2.1.1-15-2015-00370.

References

[1] "Magic Software Enterprises," http://www.magicsoftware.com, last visited May 2017.

[2] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[3] C. Krueger, *Easing the Transition to Software Mass Customization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 282–293.

[4] C. Kästner, A. Dreiling, and K. Ostermann, "Variability Mining: Consistent Semi-automatic Detection of Product-Line Features," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 67–82, 2014.

[5] W. K. G. Assunção and S. R. Vergilio, "Feature location for software product line migration," in *Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC '14*. New York, New York, USA: ACM Press, 2014, pp. 52–59.

[6] H. Eyal-Salman, A.-D. Seriai, C. Dony, and R. Al-msie'deen, "Recovering traceability links between feature models and source code of product variants," in *Proceedings of the VARiability for You Workshop on Variability Modeling Made Useful for Everyone - VARY '12*. New York, New York, USA: ACM Press, 2012, pp. 21–25.

[7] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, and I. Kovács, "MAGISTER: Quality Assurance of Magic Applications for Software Developers and End Users," in *26th IEEE International Conference on Software Maintenance*. IEEE Computer Society, Sep. 2010, pp. 1–6.

[8] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, and I. Kovács, "Solutions for reverse engineering 4gl applications, recovering the design of a logistical wholesale system," in *Proceedings of CSMR 2011 (15th European Conference on Software Maintenance and Reengineering)*. IEEE Computer Society, Mar. 2011, pp. 343–346.

[9] J. V. Harrison and W. M. Lim, "Automated Reverse Engineering of Legacy 4GL Information System Applications Using the ITOC Workbench," in *10th International Conference on Advanced Information Systems Engineering*. Springer-Verlag, 1998, pp. 41–57.

[10] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 125–135.

[11] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, and I. Kovács, "Complexity measures in 4gl environment," in *Computational Science and Its Applications - ICCSA 2011, Lecture Notes in Computer Science*, ser. Lecture Notes in Computer Science, vol. 6786. Springer Berlin / Heidelberg, 2011, pp. 293–309.

[12] D. Falessi, G. Cantone, and G. Canfora, "A comprehensive characterization of NLP techniques for identifying equivalent requirements," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*. New York, New York, USA: ACM Press, 2010, p. 1.

[13] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society of Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[14] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[15] J. Zsibrita, V. Vincze, and R. Farkas, "magyarlanc: A Toolkit for Morphological and Dependency Parsing of Hungarian," *Proceedings of Recent Advances in Natural Language Provessing 2013*, no. September, pp. 763–771, 2013.

[16] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The Impact of Classifier Configuration and Classifier Combination on Bug Localization," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1427–1443, oct 2013.

[17] J. Verner and G. Tate, "Estimating Size and Effort in Fourth-Generation Development," *IEEE Software*, vol. 5, pp. 15–22, 1988.

[18] G. Witting and G. Finnie, "Using Artificial Neural Networks and Function Points to Estimate 4GL Software Development Effort," *Australasian Journal of Information Systems*, vol. 1, no. 2, pp. 87–94, 1994.

[19] "Homepage of Magic Optimizer," http://www.magic-optimizer.com, last visited May 2017.

[20] "Homepage of M2J," http://www.magic2java.com, last visited May 2017.

[21] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "Reverse Engineering Feature Models from Programs' Feature Sets," in *18th Working Conference on Reverse Engineering*. IEEE, oct 2011, pp. 308–312.

[22] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, New York, USA: ACM Press, 2011, p. 461.

[23] R. Al-msie'deen, A.-D. Seriai, M. Huchard, C. Urtado, and S. Vauttier, "Mining features from the object-oriented source code of software variants by combining lexical and structural similarity," in *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*. IEEE, aug 2013, pp. 586–593.