Feature Analysis using Information Retrieval, Community Detection and Structural Analysis Methods in Product Line Adoption

András Kicsi, Viktor Csuvik, László Vidács, Ferenc Horváth, Árpád Beszédes, Tibor Gyimóthy

Department of Software Engineering and MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, Hungary

{akicsi,csuvikv,lac,hferenc,beszedes,gyimothy}@inf.u-szeged.hu

Ferenc Kocsis

SZEGED Software Ltd., Szeged, Hungary, kocsis.ferenc@szegedsw.hu

Abstract

In industrial practice the clone-and-own strategy is often applied when in the pressure of high demand of customized features. The adoption of software product line (SPL) architecture is a large one time investment that affects both technical and organizational issues. The analysis of the feature structure is a crucial point in the SPL adoption process involving domain experts working at a higher level of abstraction and developers working directly on the program code. We propose automatic methods to extract feature-to-program links starting from very high level set of features provided by domain experts. For this purpose we combine call graph information with textual similarity between code and high level features. In addition, in depth understanding of the feature structure is supported by finding communities between programs and relating them to features. As features are originated from domain experts, community analysis reveals discrepancies between expert view and internal code structure. We found that communities correspond well to the high level features, with usually more than half of feature code located in specialized communities. We report experiments at two levels of features and more than 2000 Magic 4GL programs in an industrial SPL adoption project.

Keywords: software product line, feature extraction, information retrieval, community detection

1. Introduction

Maintaining parallel versions of a software satisfying various customer needs is challenging. Many times the clone-and-own solution [1] is chosen because of short term time and effort constraints. As the number of product variants

increases, a more viable solution is needed through systematic code level reuse. A natural step towards more effective development is the adoption of product line architecture [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. When 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 [3]. An advantage of the extractive approach in general is that several reverse engineering methods exist to support feature extraction and analysis [4, 5, 6].

We report on an ongoing product line adoption project where the new architecture is based upon an existing variant and the specific functions of the others are being merged into the final architecture (while the whole process exceeds the scope of this work). In principles it is closest to the extractive approach. Our subject is a legacy high market value, wholesaler logistics system, which was adapted to various domains in the past using clone-and-own method. It is developed using a fourth generation language (4GL) technology, Magic [7], and in this project the product line architecture is to be built based on an existing set of products developed in the Magic XPA language. Although there is reverse engineering support for usual maintenance activities [8, 9], the special structure of Magic programs makes it necessary to experiment with targeted solutions for coping with features. Furthermore, approaches used in mainstream languages like Java or C++ need to be re-considered in the case of systems developed in 4GLs. For instance, in the traditional sense there is no source code, rather the developer sets up user interface and data processing units in a development environment and the flow of the program follows a well-defined structure.

In the focus of our work is the feature identification and analysis phase of the project. This is a well studied topic in the literature for mainstream languages [5], but the same for 4GL is less explored. The method starts from a very high level set of features provided by domain experts, and uses information extracted from the existing program code. The information retrieval (IR) approach to feature extraction as a single method turned out to be noisy in Magic 4GL system analysis [10]. Hence the extraction is performed by combining and further processing call graph information on the code with textual similarity (IR) between code and high level features. Essentially the method is working simultaneously with structural (syntactic) and conceptual (text based) information, similarly that have been previously proposed for traditional object oriented systems [11]. While most related literature deals with object oriented systems, our goal is to aid the product line adoption of an existing 4GL system. This does not only bring a distinction from the language perspective but also a less general and more industrially motivated viewpoint. While similar methods can be used as in object oriented environments, these often have to be modified. IR based methods have less problems with dealing with the different paradigms but with structural information it can be difficult to achieve the same results as with a more straightforwardly structured other system. Besides combining tex-

tual extraction with structural one, we present an efficient method for filtering the data from both sources as well. This results in a set of information that is more suitable for performing the SPL adoption by various stakeholders of the project including domain experts and architects. In summary, the contributions of this paper are the following:

- A method for feature extraction by combining syntactic and textual information and using filtering results of the two sources.
- Feature analysis methods by applying community detection algorithms to match features with call graph communities.
- Application of the approach in an industrial setting in 4GL environment during a product line adoption project.

The main goal of our work is to aid our industrial partner in its current product line adoption task. While our results may have the potential to be used in case of other 4GL languages or other feature extraction work, we do not attempt to introduce a completely foolproof and fully versatile approach for every situation. The paper is structured as follows. Section 2 introduces our current task and defines our approach to tackle the problem. Section 3 provides more detailed information on our methods for feature extraction and presents our various result sets. Section 4 describes our experiments on call graph communities. Since we aim to facilitate work an analysis of the project's progress can be found in Section 5. We evaluate our various feature extraction outputs and the community detection's information value in Section 6 with the help of two research questions that we seek answers to. We overview the related work in Section 7 and conclude our paper in Section 8.

2. Feature Extraction and Abstraction of Magic Applications

2.1. Product line adoption in a clone-and-own environment

The decision of migrating to a new product line architecture is hard to make. Usually there is a high number of derived specific products and the adoption process poses several risks and may take months [12, 13, 14]. The subject system of our analysis is a leading pharmaceutical wholesaler logistics system started more than 30 years ago. Meanwhile almost twenty derived variants of the system were introduced at various complexity and maturity levels with independent life cycles and isolated maintenance. Our partner is the developer of market leading solutions in the region, which are implemented in the Magic XPA fourth generation language.

This work is part of an industrial project aiming to create a well-designed product line architecture over the isolated variants. The existing set of products provide an 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.

The 4GL environment used to implement the systems requires different approaches and analysis tools than today's mainstream languages like Java [8, 15]. For example, there is no source code in its traditional sense. The developers work in a fully fledged development environment by customizing several properties of programs. Magic program analysis tool support is not comparable to mainstream languages, hence this is a research-intensive project.

The feature extraction process is challenged, since the 19 product variants themselves are written in 4 different language versions, Magic V5, Magic V9, UniPaaS 1.9 and XPA 3.x. In case of the oldest Magic V5 systems, there is a high demand on the migration to a newer version. UniPaaS 1.9 introduced huge changes in the language by using the .NET engine for applications. Most systems are implemented in that version. The newest Magic XPA 3.x line of the language lies close to the uniPaaS v1.9 systems. Each variant is between 2000 and 4000 Magic programs in size with a large amount of code in common. Magic is a data-intensive language, which is clearly reflected by the program code as well, the variants containing 822 models and 1065 data tables in maximum.

2.2. Feature extraction approach

During product line adoption's feature extraction phase various artifacts are obtained to identify features in an application [16]. This phase is also related to 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.

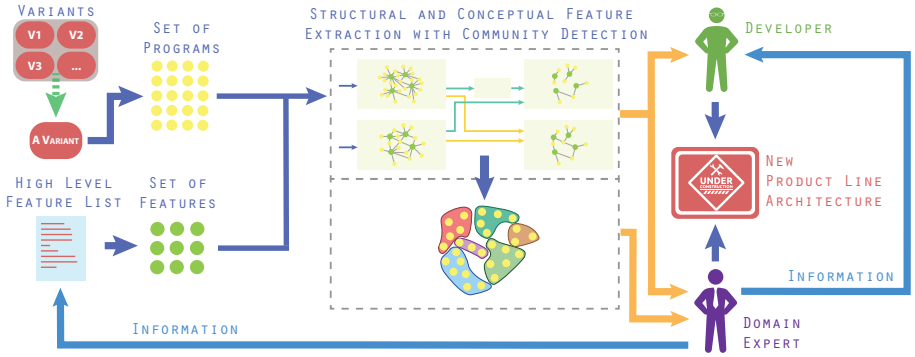

Figure 1: An illustration of feature extraction and analysis as part of product line adoption

In this part of the research project, we aim at feature extraction and analysis. Our inputs are the high level features of the system and the program code. We apply a semi-automated process as in the work of Kastner et al. [4]. High level features are collected by domain experts from the developer company. The actual task is to establish a link between features and main elements of the

Magic applications. Although there exists a common analysis infrastructure for reverse engineering 4GL languages [17, 8, 9], the actual program models differ.

Figure 1 illustrates our current approach to feature extraction. We assign a number of elements for each high level feature, this information helping the work of developers and domain experts working on the new product line architecture. This is the feature extraction phase which can be seen at the center of the figure with the green feature nodes and the yellow program nodes are organized in graphs with their connections shown. During the assignment, we mainly rely on structural information attained on call dependency by constructing a call graph of the programs of a variant. This results in a high number of located elements, crucial for the development of product line architecture, but the large amount of data can be hard to grasp in its entirety. We combine this method with information retrieval, which can also make it easier to cope with a 4GL language by utilizing conceptual connections, and is successfully applied in software development tasks, such as in traceability scenarios for object oriented languages [18]. A comprehensive overview of Natural Language Processing (NLP) techniques – including Latent Semantic Indexing (LSI), the technique we chose – is provided by Falessi et al. [19]. In our previous work [10] we already presented our LSI-based approach to feature extraction. LSI is already known to be capable of producing good quality results combined with structural information [11].

To further assist the work of domain experts, we analyze communities based on the call graph of the entire system. This opens the way to compare the view of domain experts on features with code level communities that reflect the implementation of those. Community detection algorithms address large networks and have been used for software engineering problems like handling dependencies [20] and support modularization [21], but we are not aware of their application in product line research.

2.3. The structure of a Magic application

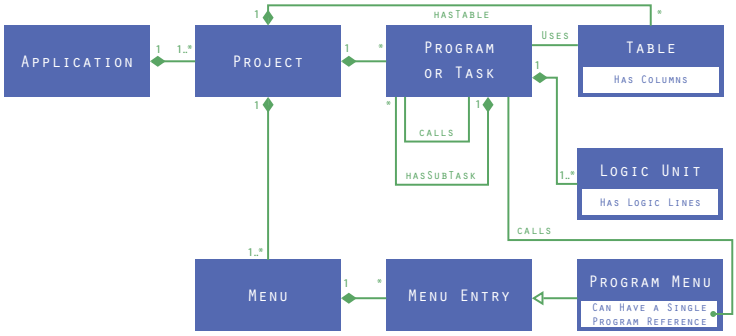

Figure 2: Illustration on the elements of a Magic application

Being a fourth generation language, Magic does not completely follow the structure of a traditional programming language. It has a lot of different ele-

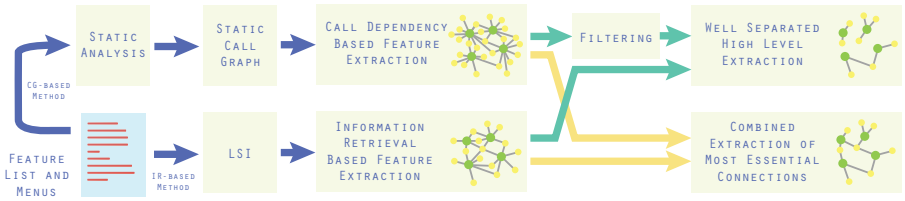

Figure 3: A more detailed view on the feature extraction process

ments, in this subsection we seek to introduce the only ones necessary for the understanding of our approach. Figure 2 aims to present these elements.

A software written in Magic is called an application. These can be built up from one or more projects. In turn, each project can have any number of programs which contain the actual logic of the software. The tasks branching directly from a project are called programs. These can have their own subtasks and be called anywhere in the project like methods in a traditional programming environment, but their subtasks can only be called by the (sub)task containing them. Any task or subtask can access data through tables.

It is also possible for a program to be called through menus, which are controls designed to provide user intervention and usually start a process by calling programs. Having sufficient information on menus, we used these as a base for the call graph in structural feature extraction, deriving calls from menus.

3. Feature Extraction Experiments

3.1. Overview

In this section, we present the feature extraction methods used based on call dependency and textual similarity, as well as the combination and possible filtering options. Figure 3 illustrates the processes described in this section. Our static analysis is specific to the Magic language. One of the variants of our subject system was selected by domain experts to be used as a starting point for the product line adoption. It is a specific variant involving 4251 programs, 822 models and 1065 data tables. The new product line was started from this variant, the capabilities of the other variants are being built into the product line during the adoption process. We have been provided with a feature list structured in a tree format initially consisting of three levels which have 10, 42 and 118 unique elements respectively. This list was being refined in an iterative manner. From these we chose the upper level to display our results, the features of this level are listed in Figure 4. The numbers shown here are in accordance with the numbers we present in our later graph examples.

1 − Manufacturing	6 − Administrator interventions
2 − Interface	7 − Supplier order management
3 − Access management	8 − Invoicing
4 − Quality control	9 − Master file maintenance
5 − Stock control	10 − Customer order reception

Figure 4: The higher level features of the system

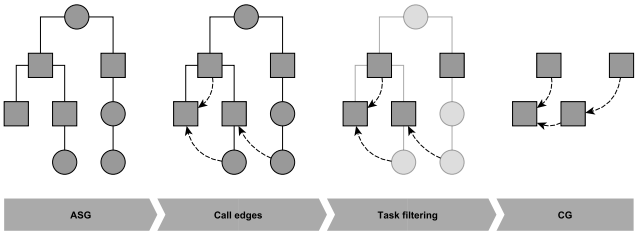

Figure 5: The process of calculating the call graph

3.2. Approach

3.2.1. Feature extraction using (task) call dependency

This approach relies on the call dependencies between programs and tasks. To construct a call graph from these dependencies we use the process illustrated in Figure 5. The figure represents a minimalistic example of a Magic application. Squares mean tasks and programs, while other program elements like projects, logic units, logic lines, etc. are shown as circles. From the source code we construct the abstract semantic graph (ASG), which is provided by our static source code analyzer tool. As the next step, we add the call edges to the graph by examining Magic elements that operate as calls between tasks and programs. Finally, in the last two steps of the process we eliminate some nodes and edges from the graph, keeping only the necessary ones *i.e.,* call edges, tasks, and programs. From the CG we obtain the features by running a customized breadth-first search algorithm from specific starting points determined by menu entries. The domain experts assembling the feature list know the menu structure well, hence we considered the given feature-menu connections a good starting point for call graph construction. You can see a graph representation of the CG based results later on the left side of Figure 7.

3.2.2. Textual similarity

By the textual nature of information retrieval (IR) techniques, they are less susceptible to various hardships some other techniques face like the language used. In our case, one of the biggest problems is that the systems processed use different versions of the Magic language. Thus IR can contribute to a more versatile approach. For this purpose, we used the Latent Semantic Indexing technique [22] for measurement of textual similarity. A more complete summary of our feature extraction work with LSI is presented in our previous work

[10]. Similarly to the CG technique, we also used information retrieval for determining connections between features and programs of the system. Though the structural information obtained from call graph is more thorough, it is more suitable for the developers rather than domain experts. With purely structural information it is hard to separate along the features, having many programs laying the groundwork for any single feature it is hard to grasp the overall aim. This conceptual analysis separates more agreeably along the semantics of the feature, hence it can be more valuable for domain experts. A graph representation of textual similarity results can be seen later on the right side of Figure 7.

3.3. Results and Further Possibilities

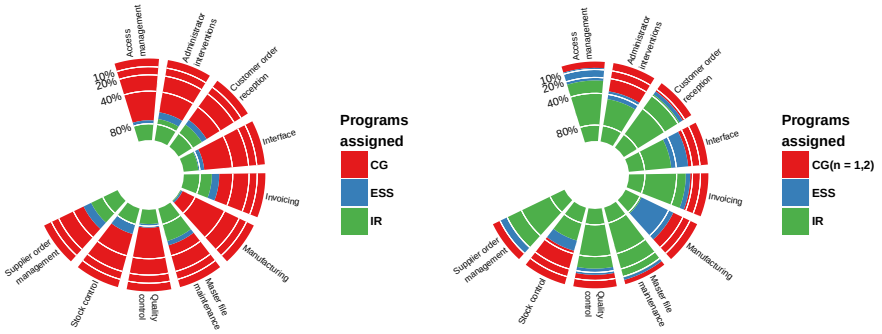

Figure 6: The size of our result sets for each feature with each technique. Results without filtering shown on the left, results with filtering on the right. (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

As already introduced, the two methods we used for program assignment to features use fundamentally different methods for achieving their results. Consequently, the results themselves also show a significant difference, overlapping only partially. The set of programs for the techniques presented are shown on the left side of Figure 6. Each slice of the diagram represents a top level feature and its colors indicate the number of programs detected by each technique. IR represents the result set of the information retrieval technique, CG represents the pairs attained by call graph, while ESS represents the set of programs considered most essential, detected by both techniques. The left side of Figure 8 shows the number of programs assigned in each set, the abbreviations match the ones explained for the previous figure.

The call graph dependency technique provides a high number of programs for each feature as displayed on the left side of Figure 7. These relations are based on real calls of the code itself which can be considered a reasonable source of information. It is important to note that we only used static call information, thus at runtime not every call occurs inevitably. Developers need to work with programs, hence they are required to have some readily available information on all of them. The call information which this technique uncovers is likely

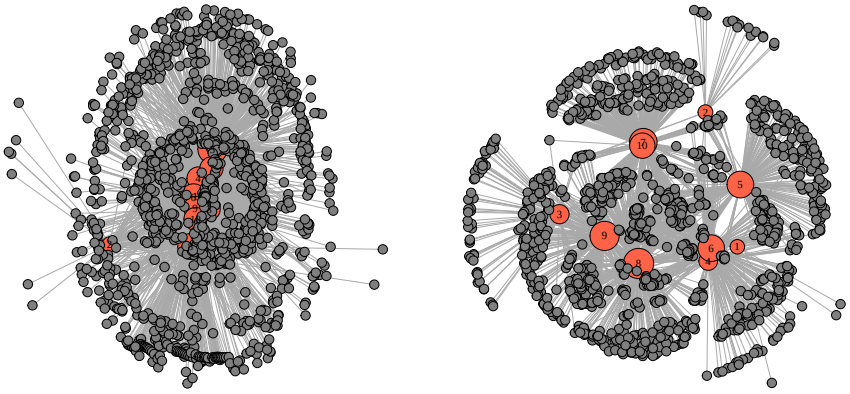

Figure 7: Graph visualization of the set of results obtained by the call graph (Left) and the information retrieval (Right) technique

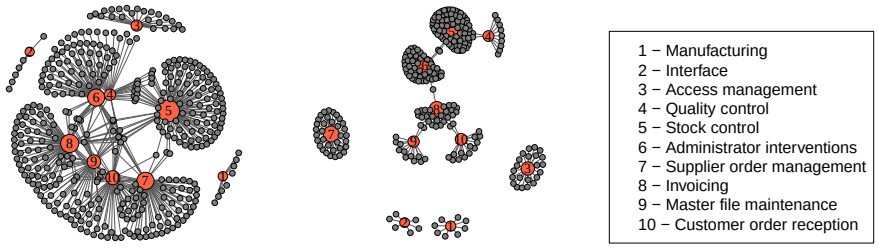

Figure 8: Graph visualization of the set of programs deemed most essential. Results shown on the left, results with filtering on the right

to benefit their basic understanding of the programs, but it also presents a problem of coping with the large amount of data not really distinguishable in any manner.

The conceptual method produces fewer programs for each feature as can be seen on the right side of Figure 7 where we show a graph representation of the IR-based results. Further examination of random cases revealed that even considering this, a significant amount of noise presents itself. Textual similarity works with very little information in these cases, hence it is likely for similar wording or more general words like "list" to produce misleading matches, occurring in the text of many features. So this method can connect radically different parts of a system both in the aspect of features and the system structure, hence we consider this conceptual method less precise.

Looking at only the intersection of the connections found by these two techniques we find that this set of connections takes into account both the struc-

tural and conceptual information, producing only connections which are indeed present on both levels. This results in a clearer, more straightforward set of connections, which contains the most essential findings of the two techniques.

As we could see before, the structural information produces a rather large amount of matches for each feature, and we observed that there is a considerable overlap between features. We decided to attempt to clear these matches too with a filtering technique applied on the structural information output, which filters out less specific programs. The filtering technique works with a number n, which denotes the maximal number of features a program can connect before it is considered less specific and is filtered out from the program set of features. This removes the programs with less information value and results in even more straightforward groups of programs for each feature. We have to note that less specific programs are often no less important, but their relations are harder to comprehend, which is our focus in the current case. On the right side of Figure 6 and Figure 8 we can see the results of the common structural and conceptual connections of this filtered approach, featuring only programs with maximum two connections. It is apparent from the graph that features are much better separated, providing a suitable high level glance at the background of features without a lot of technical details, ideal for top level understanding.

Examining the graphs we can come to many interesting conclusions. For example feature number 7 is behaving like any other feature considering the purely conceptual or purely structural viewpoint, its common graph provides a clearer picture, apparently connecting through a group of more general features to a large number of other features. In the filtered case however, it is nicely separated with a group of unique programs specific to the feature itself.

4. Feature Analysis using Call Graph Communities

4.1. Overview

Community detection provides a grouping of programs with denser inner connections. As already established, feature detection also results in a grouping of the programs, just as community detection. This means that we can compare these two sets of results in a relatively easy way. This comparison can be useful in many ways enabling us to reach new information about the feature model and even to refine the output of the feature extraction process itself. In this section, we present our experiments with various community detection algorithms and investigate their possibilities in aiding product line adoption when combined with the previously constructed outputs of feature extraction.

4.2. Approach

Communities are groups of nodes located in a graph. They are more densely connected internally than to the rest of the graph. They are often researched topics of network science and contribute highly to scientific and industrial research. Groups of densely connected nodes often exist in almost every network. Since communities are not strictly defined, several different correct groupings

of graph nodes can exist. There are a large number of community detection algorithms used, often even specialized to a specific field, like social networks.

In our current research, we consider the programs of the system as the nodes of the graph and do the community detection on them. The edges of the graph are the calls the programs make extracted by static analysis. The detected features are not indicated in the graph and play no part in the community detection process. As the call edges are more dense inside the communities, we can expect that these programs work together more closely and thus perform similar tasks. This is exactly what characterizes features too, so we would expect significant overlaps in communities and detected features.

There is a key value in community detection, modularity, which is a scale value between -1 and 1 that represents the density of the edges inside communities compared to the edges outside communities. Theoretically, optimizing this value results in the best possible groups of nodes. Computing this completely is rather complex, thus in practice we rely on different heuristic algorithms. During our experiments we considered the following community detection algorithms provided by the R software environment:

Edge Betweenness: Edge betweenness scores are the number of shortest paths that pass through an edge. The algorithm removes the edges by a decreasing order of these scores.

Fast Greedy: Each node starts as an individual community, and these are merged together in a locally optimal manner considering the largest increase in modularity.

Leading Eigenvector: In each step the graph is split into two parts in a way that maximizes modularity. This is determined by the leading eigenvector of the modularity matrix computed on the graph.

Louvain: A multi-level algorithm, consists of repetitions of greedily assigning locally optimized small communities and then considering these assigned groups as individual nodes.

Walktrap: Starts out random walks from the nodes, and because these tend to leave communities less often, determines community merges according to them. It has a parameter t which represents the number of steps in each random walk.

To quantify the results of these community algorithms, we decided to use some metrics that can contribute to the understanding of the output. One obvious metric is the number of communities assigned. Different algorithms produced a varying number of communities on the same data. For easier handling we experimented with changing this circumstance and demanding a previously defined number of communities. This approach did not do well since it often produced a large number of tiny and some oversized communities which separated the graph very badly and this provided no real information value. We computed the following metrics for further investigation:

NoC: Number of communities.

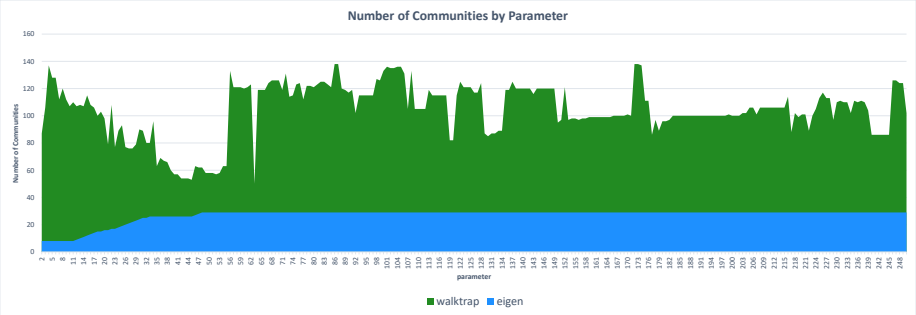

Figure 9: Number of communities at various parameter settings of the Walktrap and Leading Eigenvector methods

MFC: The maximal coverage of a single feature.

NoCSC: Number of communities with significant coverage. We consider coverage significant if the community covers at least 10% of the programs of a feature.

AFC: The average percentage of programs of features covered by communities with at least 10% coverage

4.3. Results

4.3.1. Comparison of Community Algorithms

Algorithm	NoC	MFC	NoCSC	AFC
Edge Betweenness	123	31.81%	6	45.84%
Fast Greedy	44	31.81%	7	57.09%
Leading Eigenvector	29	45.35%	9	61.75%
Louvain	32	36.36%	10	52.91%
Walktrap (t=40)	57	40.91%	6	56.63%

Table 1: Results of our analysis with the five community algorithms for top level features

Table 1 presents the results of the metric values measured on the top level of features. Each row represents the results of a different community algorithm. For Walktrap we have chosen a *t=40* walk length which according to the results of different experiments provided the most suitable number of communities. The Leading Eigenvector method also works with a parameter, in this case, we have chosen *29* for similar reasons. Figure 9 shows the number of communities at each value of the parameters of these techniques listed from 2 to 250. A large number of communities can present too much granularity especially for comparison with the top level of features since we have only ten features on this level. Having a greater number of features can make the large number of communities beneficial. The results of the table indicate that Edge Betweenness

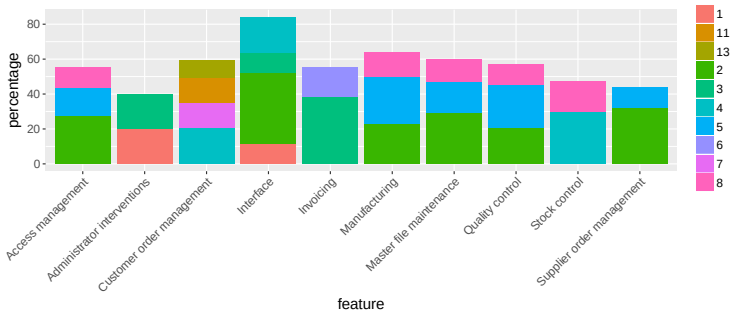

Figure 10: The proportions of communities determined in case of each feature at top level with Walktrap (t=40) detection and a 10% feature coverage filtering

detects a significantly larger amount of communities than the other algorithms. These are usually very small with a few larger communities. The MFC value represents the largest coverage of a single feature, meaning that for example in the Edge Betweenness case there was a community which covered 31.81% of a single feature, which was the largest value in this scenario. As we can see this number can vary greatly through different algorithms. This number however only describes the coverage of a single community rather than a representation of all. NoCSC, on the other hand, provides an exact number on this property, representing the number of communities that cover at least 10% of a feature. As we can see this is not a large value in any case. The lowest values were achieved by the algorithms with the highest NoC values, which is not surprising since these qualities can be somewhat opposite. Since there is more granularity, the smaller groups are bound to cover less from each feature. AFC represents the total coverage achieved by each feature on average considering communities with at least 10% of a feature covered. As we can see this number moves around 55% with Edge Betweenness differing significantly, which is most probably also due to the high granularity. This means that on average more than half of the programs of a feature are located in communities containing a significant part of that feature. It can suggest that the main functionalities of the features are mostly achieved in communities specialized to that single feature or some larger communities which cover an important part of several features. Figure 10 illustrates the rate of coverage for each feature by communities that represent at least 10% of the feature's activity.

4.3.2. Feature analysis using communities

Domain experts can play a significant part in the adoption of a new product line architecture. To their work complete knowledge of the features is invaluable. Aiding this, community detection can highlight previously unknown aspects of the systems. As the community detection is based on actual call edges of the call graph, the programs attached to each other are indeed meant to be interacting during the proper run of the system. With this knowledge, we can assume that the programs working towards the same goal should have more

calls to each other, which makes communities an ideal and easy way to form groups of these programs. Possessing knowledge of groups of programs inside the system can also aid the testing process greatly. Testing the product line usually tests functionalities the system provides, with other words it tests the features themselves. While the features are identified with proper feature extraction, testing is still done on the code itself. As a consequence of this, testing a single feature can involve programs that are not foreseen by domain experts. Community detection can provide a way for domain experts to form realistic expectations of the involved parts of the system thus representing an additional valid and not overly complex perspective about the system.

As mentioned before, we have experimented with five different community algorithms. Of these we have chosen Walktrap because it produced a manageable amount of communities which still seemed relatively unanimous. We remark that all community detection algorithms produced similar results and we did not see any significant differences that could not be mainly caused by simply the number of communities identified. One important advantage of Walktrap can be that it can produce a variable number of communities. Changing its parameter we can get more, which can improve the situation on lower feature levels where more features are present which are better represented with more smaller communities rather than a few medium sized ones. A graph illustrating top level feature results with Walktrap communities can be found in the appendix in Figure A.22, we will also analyze some of these in the current section. Each node represents a program, the edges are the calls the programs make according to the call graph. The colors of nodes represent the features that they were assigned to in the call graph based feature extraction. This display is not complete since there are programs with more than one features and we could only color the nodes according to one of these. However, the names of all assigned features appear on the nodes, so there is no information loss. Communities are represented by the location of the nodes, each community forms a circle of its programs. We use these same notations in all the following community examples.

Generally, we can say about the whole graph that various communities are present. As with every community detection method, we can observe some really large communities that usually involve programs of a lot of different features. On the other hand, we can see many smaller ones with less variance. For example, the three communities presented in Figure 11 are not particularly large but each can be considered as belonging mainly to a single feature. It is apparent that the left community is part of Administrator interventions, the central community belongs to Supplier order management, as does the community on the right but it also takes part in the work of several other features. Orange nodes tend to be more general in the whole graph, supporting a lot of features, with only four communities having programs that only deal with Access management. This can mean that Access management is usually more interwoven with other features. The same can be said about Manufacturing with an even higher extent. Administrator interventions, on the other hand, owns a great number of communities exclusively. While small differences inside a community

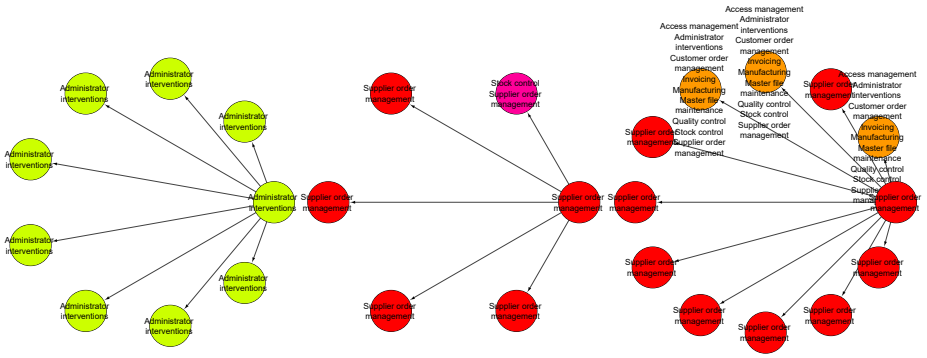

Figure 11: Three communities with well distinguishable goal. Each node represents a program of the system, while its color marks its feature affiliation. See the node text for cases of more than one features assigned. The edges are the calls the programs make while the circular groups represent their assigned communities. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

can be present, each community detection algorithm produces a high number of communities that are very similar to these, centered around different features. In the current level, almost every top level feature has at least one community it is very dominant in except two of the features, Manufacturing and Master file maintenance. Interface only seems to appear in communities with a lot of Access management programs which can suggest its heavy reliance on Access management.

In Figure 12 we can see another example taken from the graph. We can see a large community which consists of programs of many features and also a medium sized one with two dominant features, making several calls to programs of the large community. The figure only contains the edges that have both endpoints in these communities but it is already apparent that there are two programs inside the large community which are targeted by a vast amount of program calls. If we take a look at the full picture in the appendix we can see that there are still many more call edges to these programs from various communities. Since these programs represent a lot of features this suggests that they are some of the most essential programs that almost every feature relies on. They can also be the cause for the detection of such a large community since a lot of programs rely solely on them. It is also interesting that while targeted by a lot of calls they do not seem to make any. This implies that they provide some service that is routinely needed rather than play a vital controlling part.

We can also see some gray nodes that feature extraction did not assign any features to. This can potentially mean abandoned or not yet used code or simply a result of the imperfectness of a feature extraction process. It is not apparent from the graph examples but these nodes do serve as an endpoint to a call edge since programs that neither make or are targeted by calls do not appear. Communities could also extend feature extraction outputs since if we encounter featureless programs inside a community with a highly dominant feature we can

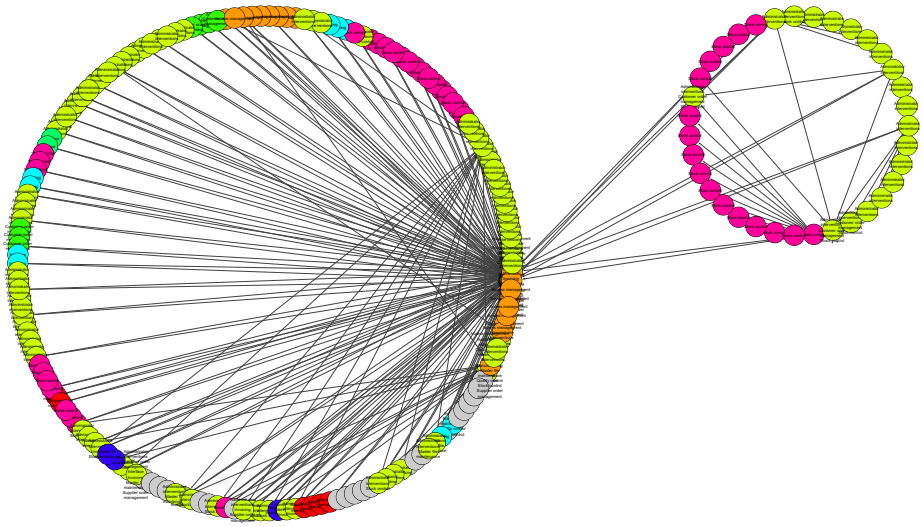

Figure 12: A larger, more general community involving a lot of features and a medium one with two main features. Each node represents a program of the system, while its color marks its feature affiliation. See the node text for cases of more than one features assigned. The edges are the calls the programs make while the circular groups represent their assigned communities. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

suspect that it serves the same feature.

Applying community detection algorithms on the programs of a Magic application we have found that the communities overlap with our feature extraction output. Apart from a few large, more general communities, we can detect several clusters of programs that are specialized to achieve a single feature. Our investigation showed that the majority of top level features are represented by at least one community of their own. While sometimes different specialized communities also arise or some disappear, the same conditions can be observed with all 5 community detection algorithms we have involved in the experiment.

4.3.3. Analysis at deeper level of features

The results presented above only involve the top level features we have been provided by domain experts. These features can be further detailed to a second feature level and the analysis can be executed here also. This can result in a more comprehensive picture. A further advantage of community detection on a lower level is that the algorithms used do not depend on feature level, and they tend to identify more communities than the number of top level features at our disposal. Usually, there are a lot of very small communities. These tend to perform subtasks of the same task and also belong to the same feature. Since they are small, it would be a fair assumption that they may represent lower level features working together for the same smaller goal, contributing to the top level feature. Making a comparison of lower level features and these

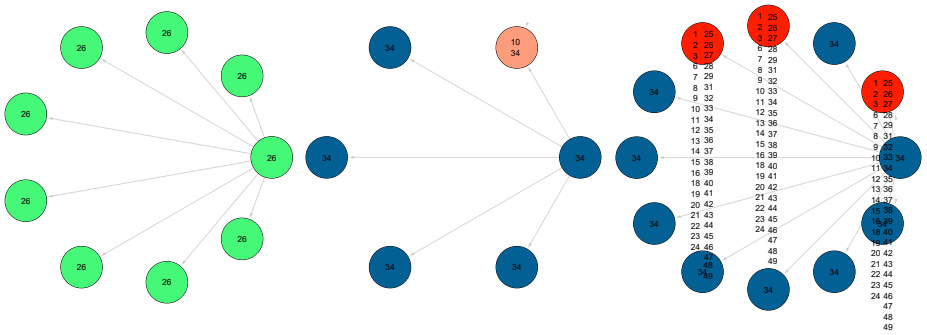

Figure 13: Three communities from second level feature extraction

smaller communities can highlight or disprove this. Additionally, there are larger communities which are likely to involve more lower level features, these can be seen as working more closely together. Thus community detection can highlight valuable information at lower feature levels about both lower and top level features alike. With this, we can gain more information about the actual inner working of the system which is worthy of examination. Level 2 features can also present more interesting information because the top level of features is often considered too general for actual work while second level features represent a more detailed picture of the actual functions the features provide. Since there are only 10 features at the top level while there are 49 on the second, community detection at this level can also lead us to valuable observations.

Results of second feature level community detection are fully available in the appendix in Figure B.23. Three of its communities are presented here in Figure 13. The feature names in these figures were omitted because of the length of their qualified names and their large quantity. Instead of their names we have displayed numbers representing their level 2 features. While a lot of programs still represent only one feature, is not rare here for one to belong to tens of features. At the second level, it is observable that specialized communities are still present in large numbers. Programs that only serve one feature tend to occur together mostly with only a few programs that also work on some additional features. The communities presented in the example are actually the same as the ones we have seen in Figure 11, now with the second level of features. Apparently, no significant changes have occurred. From one aspect this is not surprising since the call edges themselves did not change, only the feature classification of programs. This means that the communities are bound to remain the same, only consist of programs belonging to different features. The programs belong to the same top level feature as before, now assigned to a more specific subfeature. This means that adopting a more specific level of subfeatures did not change the inner variability of these communities. The same circumstances can be observed in general, most of the specialized communities retained this quality.

On the other hand, two major differences can be seen from top level. One

of these is that programs belonging to a large number of top level features seem to have gained even more subfeatures in this scenario. This means that these programs take part in even more features. The interesting aspect of this is that there seems to be no middle ground, a program either supports just a selected few features or almost every second level feature. According to this, we could easily divide programs into two categories, specialized and general programs.

The other difference is that the large communities seem to have gained even more variance, they contain a significantly higher amount of different features which means that their features consist of more subfeatures. Considering their size this is not surprising yet now it is apparent that while these subfeatures rely more on programs of other subfeatures, the programs of the more specialized communities tend to work mostly in separation. This can be an important distinction and can provide information of value in many cases. For example, if we are contemplating a change, we can see how much unintended impact that change can have for each subfeature. Specialized communities tend to be more isolated, hence they can be more easily changed or turned off as a feature. This can be crucial both for product line adoption and for maintenance reasons.

Adopting a deeper level means that the number of features increases significantly which could potentially divide specialized communities and produce more general ones. As we have found this is rarely the case. These communities seem to retain their good quality of being largely dominated by a single feature even on lower levels. As the number of features rose from 10 to 49 we experienced no significant change in the rate of specialized versus more general communities. This can even verify the feature model itself since the programs belonging together according to the feature list seem to also work more closely together in reality.

5. Insights Into the Progress of SPL Adoption

In this section, we provide information of how the project progressed through time and examining the effects of our work. The system and the feature list are constructed in an iterative manner, hence we can compare their versions. We have several versions of both at our disposal, with 4 separate versions of the system under construction and 7 stages of the feature list. We are going to refer to the system versions now as SV1 to SV4 and the feature list versions as FV1 to FV7.

Let us look at the changes made on the feature list first. The rates of feature additions and removals can be seen in Figure 14 where the circles correspond to the transitions between versions. Each circle consists of four tracks which represent the changes on their specific feature level. The levels are in an increasing order from the center, level 1 represents the broadest categorization of features, while level 4 is the most detailed view. We depicted the menus here as a fourth level since through most of our work they were manageable accordingly. As the list was being developed features got added to or removed from each level of the list. Though there is no guarantee that a new iteration necessarily produced

a better feature list, domain expertise is still the most reliable source of information in these cases. The changes presented here are the modifications the domain experts deemed necessary.

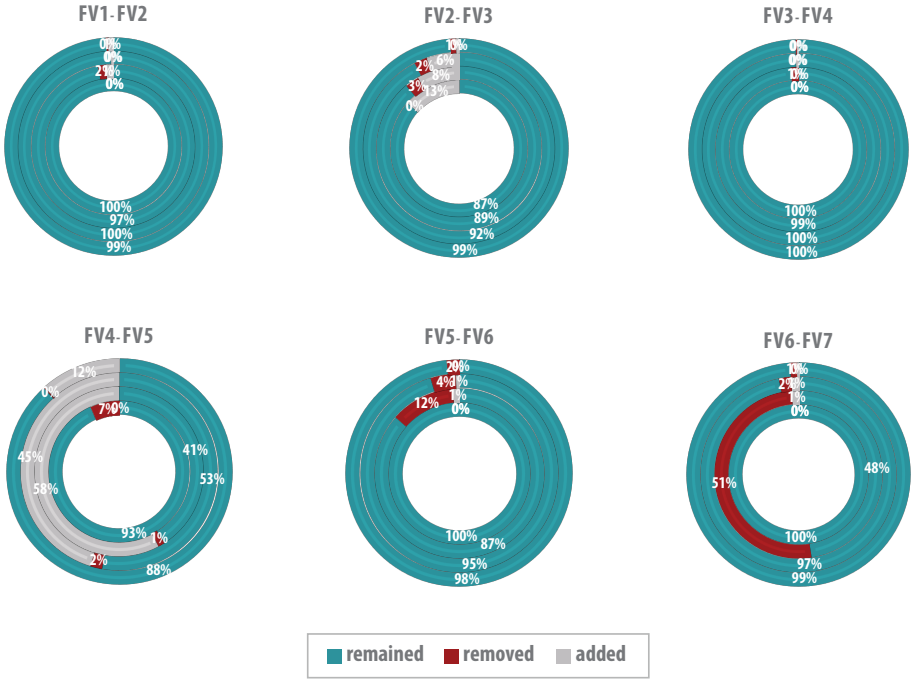

Figure 14: The changes made at each transition of feature list versions

Let us take a more detailed look at the figure by addressing some of the major changes. The FV2 to FV3 transition introduced a few new features, the most interesting of which are two new level 1 features. In the FV4 to FV5 transition, a great number of features were added, level 2 got 94 new features, level 3 got 223, while 98 new menus were introduced. This represents a major update to the feature model and the new additions make up a significant part of the new feature list. Some deletions also happened, most notably one first level feature has been removed. In the FV5 to FV6 transition, we can see the removal of 20 level 2, 20 level 3 features and 15 menus as well. The number of level 2 features was much more severely decreased in the FV6 to FV7 transition where 72 features, more than half of its total (including many of the relatively new ones) have been removed.

Apart from complete additions or removals, a significant rate of these changes stem from splitting up an original feature or merging two of them. We have also experienced several more special cases where features got "promoted" or "demoted" to another feature level, including even the level of menus. We examined these changes and have detected 9 feature promotions in total while 2 of the features got demoted. None of the features transitioned two levels or

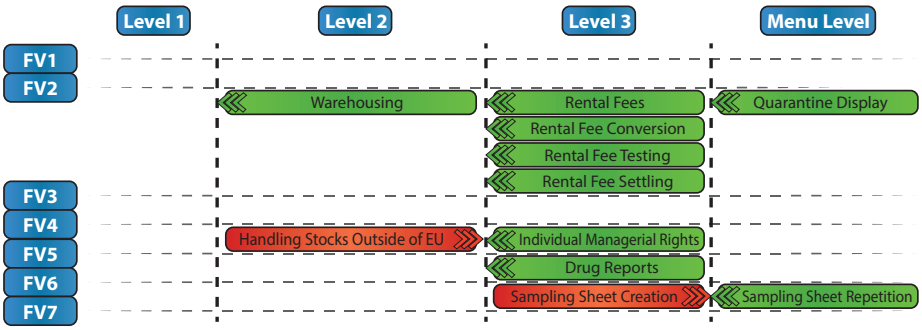

Figure 15: A summary of feature level transitions in the seven feature list versions, the green blocks with arrows on the left represent promotions while the red blocks with arrows on the right represent demotions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

moved twice. Figure 15 illustrates all of these feature movements that happened during the feature list version transitions currently referenced.

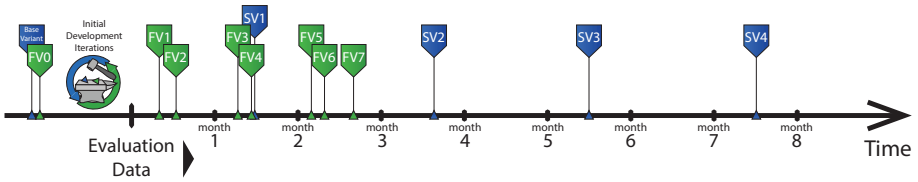

Figure 16: The timeline of created feature list and system versions currently referenced

Let us now address the versions of the system under construction. We have access to four different, relatively recent versions of the system. The versions of the feature list and the versions of the systems had different milestones, hence these two sets of data are not completely synchronized. As the building of the system itself follows the feature list, the feature list should always be considered more up to date than the current state of the system. Figure 16 illustrates the progress of the project during the time the feature list and system versions examined in this section were created. We can also see a variant depicted as base variant, this is the chosen variant the product line is being built upon. There is also a feature list version with the name of FV0, this is the feature list we did our experiments on and what we chiefly write about in the previous and future sections. There have been several other versions of both feature lists and systems, these can be seen in the figure as the initial development iterations. This period also introduced considerable changes, even the first level of the feature list was expanded with three new features. From the timeline, it is apparent that the feature list milestones followed each other quite rapidly, while the development generally took up more time.

Considering system versions, the properties of each currently referenced milestone can be seen in Figure 17. The columns represent the number of programs

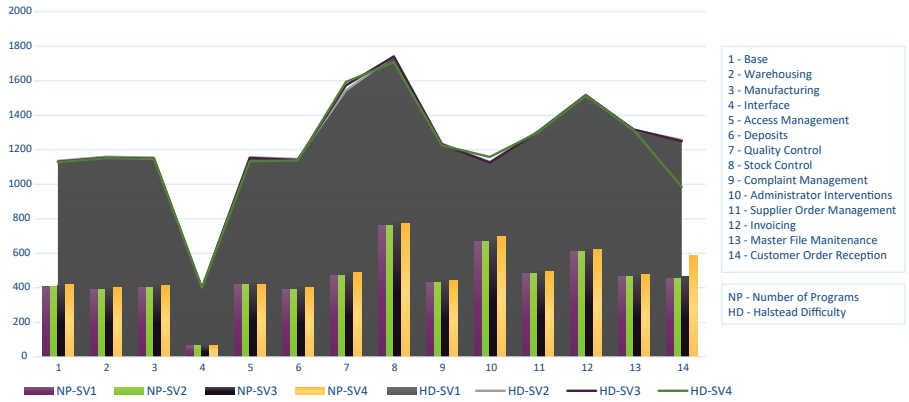

Figure 17: A comparison of the four versions of the system regarding their number of programs (NP) and their complexity (HD)

of each feature, thus display information about the size of a feature. We can see at first glance that during this time no extraordinary changes were made, but we can detect some small changes at each version transition of the system at every single feature. The largest differences present themselves at the latest version change, particularly in the case of the Customer order management feature and much less prominently at the Administrator interventions feature where it experienced a significant rise in the number of programs. This number tells us a lot about which features have been modified through time but it is not complete, since the programs themselves can be subjects to modifications as well, thus the consequences of the changes do not necessarily show up in the numbers. That is why we also displayed the complexity of these features at each version, these are represented by the lines of the figure. In our previous work [23] we have described our complexity measurement techniques regarding these features, we presented the Halstead Difficulty metric for features in Magic systems, which is basically an aggregation of the difficulty metrics of all programs of a feature. This metric basically reflects complexity as a measurement of fault sensitivity. From the figure, we can see that considering complexity the most prominent change also happened at the last version transition of the system and - like we have seen at the feature sizes - also at the Customer order reception feature where there is a major decrease in the complexity of the feature. This means that the then newly added 125 programs are generally less complex and fault sensitive than the feature's previous programs. Several other minor changes and tendencies are also visible, the size of nearly every feature seems to be in a very small growth while the complexity seems to show a very slight decrease through time in the case of most features.

As it was visible from the timeline (Figure 16), the system versions follow the feature list changes much more slowly, the development is still underway. It is nonetheless apparent that the feature list and system versions seem to be headed in the same direction. While comparing them we can see that in the

earlier versions (see Figure 4) there are several features (e.g. Feature Complaint management and Deposits at the top level) that are not even represented in the call graph of the system (see Figure 7) because they do not even make a single call. Comparing the latest examined versions however we can detect much less of these only nominally existing features. This change is certainly welcome since it indicates that the system's progression indeed follows the feature list relatively well.

6. Evaluation

In the current section, we aim to evaluate our methods by comparing our results to ones given by human experts working for our industrial partner. We involved two developers and two domain experts working on the project and asked them to fill out our questionnaires based on their expertise. These experiments were conducted retrospectively on the feature version identified as FV0 and the base variant both referenced in previous sections. For the evaluation we propose the following research questions:
RQ1: *To what extent our structural and conceptual information based feature extraction methods contribute to the correct mapping of features?*
RQ2: *What additional, not inherently available feature coupling information do communities reveal?*

6.1. Feature Extraction Outputs

For RQ1 we have devised the following evaluation process: We have chosen 18 programs at random from each of our main feature extraction outputs and also outside all of the outputs. We shuffled these programs into a single list summing up to 72 programs. This process was performed with three different randomly chosen features, thus three lists were assembled with 216 programs overall. The programs were represented both by their names and their internal identifiers so the developers could also look them up from the system itself. We asked two developers familiar with the system variants to separately judge by what extent each program is connected to the feature it was listed under. The developers were only provided with the programs listed for each of the three features and have been provided knowledge only on the possible choices they could make with no further explanation of the size or number of different program sets involved.

The results gathered with this questionnaire are displayed in Figure 18. We can see the developer opinions grouped into four separate diagrams by the three chosen features and also an overall sum of these. The first developer's impressions are displayed on the left side of each diagram while the opinions of the second developer are presented on the right. IR depicts the output of our information extraction based approach, CG represents the output based on the call graph while ESS depicts the programs extracted both by the IR and CG processes and are considered most essential for comprehension. The programs represented by the Unrelated set are chosen from outside of these other sets.

The sets contained 18 programs each and are all disjoint from each other. Maintaining these criteria the 18 programs were chosen randomly from each set. We have allowed -1 as an option for the case a developer could not come to a conclusive answer, thus there are several cases where the sum of our three differently colored columns does not come up to 18. We still have a minimum of 15 evaluated programs for each case.

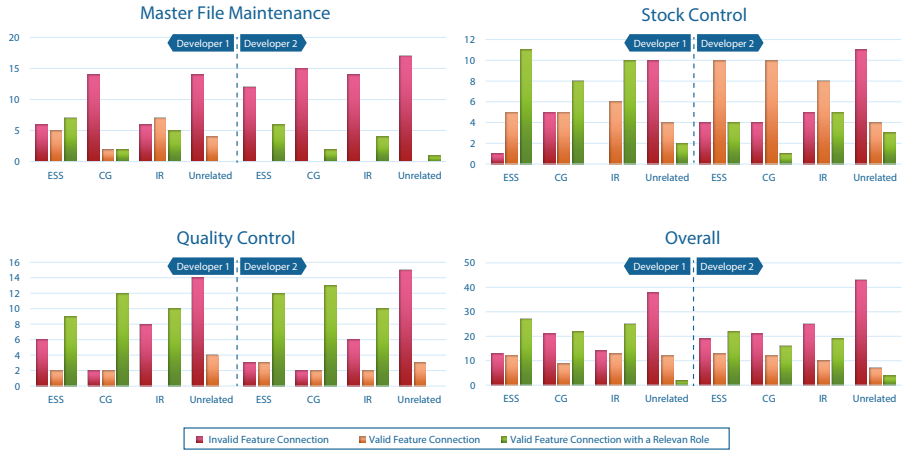

Figure 18: Evaluation of our feature extraction outputs according to two developers

It is visible that in most cases the developers found our combined results the most relevant of the sets provided, the ESS set displaying both the highest green and lowest red columns in the majority of cases. It is also clearly visible that the Unrelated set containing programs not featured in any of our outputs scored the worst in every single case. The only divergence from the success of the ESS results comes up at the Quality Control feature where their results seem to be highly surpassed by the values of CG. There are significant differences between the answers given by developers, this partly stems from their own definitions of a program being connected to a feature and being relevant. For example, it can also be seen that in general Developer 1 seems to consider the IR results significantly more valid and relevant than Developer 2.

In Figure 19 we summarize the answers the developers provided on these program sets. We can see that the developers generally considered the combined results to be the best as this set scored the least in invalid connections and scored highest in both valid feature connection categories. We can see a similar difference between CG and IR as well as IR usually performed a little better than CG, though as already noted this preference was not equal among the two developers. The Unrelated set scored the worst in every case which is not surprising but still demonstrates that our outputs hold valid information and can be used as a reference.

Answer to RQ1: Based on the evaluation results our outputs hold valid feature extraction information in overall at least 60% of matches based on the

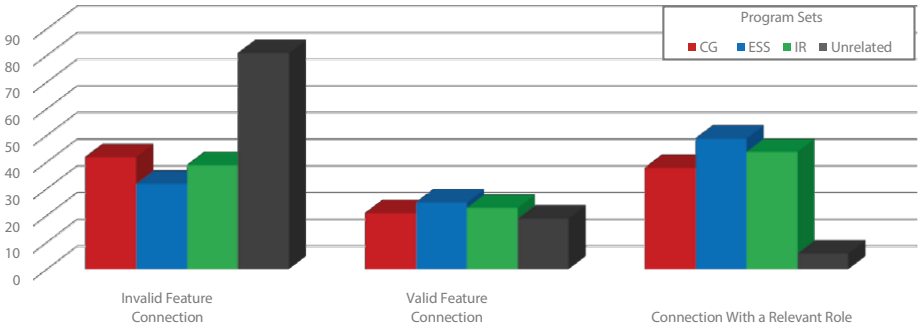

Figure 19: The sum of evaluation answers given by the developers

opinion of developers. Our output combining structural and conceptual information contains the more relevant programs of a feature, with more than 70% valid matches with over 45% also being relevant in overall. At least the third of these combined matches were relevant in every single case.

6.2. Communities

Seeking an answer to our second research question we also decided on manual evaluation. Since our community detection based output aims to contribute mainly to the work of domain experts, we asked two domain experts working on the project to provide answers regarding the mutual dependency of features. We listed the 10 high-level features introduced in Section 3 and asked the experts to fill out a table in accordance with their views on how likely it is that a change occurring in either of two features would lead to a need for the necessity of also changing the other feature. The possible answers were zero for unlikely, one for uncertain and two for certain.

Figure 20 presents the results of the questionnaire, the answers given by the two domain experts are marked red and blue. The color is darker if the domain experts have thought the dependency certain and lighter if they found it uncertain, the white fields represent the choice for unlikely.

To make the community data measurable we decided to quantify to what extent two features belong to the same communities. We computed their relative weight inside each community compared to other features and we normalized this with the size of the community. The sum of these results gives our community dependency value which can be divided into multiple categories. This is explained in the equation below where C is the set of communities, F is the feature set, w_1 is the first feature's weight, and w_2 is the second feature's weight. This resulted in a number between 0 and 3 in each case.

$$CommunityDependency = \sum_{c \in C} \left(\frac{1}{size(c)} \frac{w_1^2 + w_2^2}{\sum_{f \in F} w_i} \right)$$

The Community Connections part of Figure 20 shows how this number of each feature pair compares to the average of the opinions of the two domain experts. The black fields note the cases the community based result showed

lower connection probability level while matching connectivity levels are represented by green fields. The yellow and orange fields represent the cases where the communities suspect a higher chance of connection, the orange is a more significant difference.

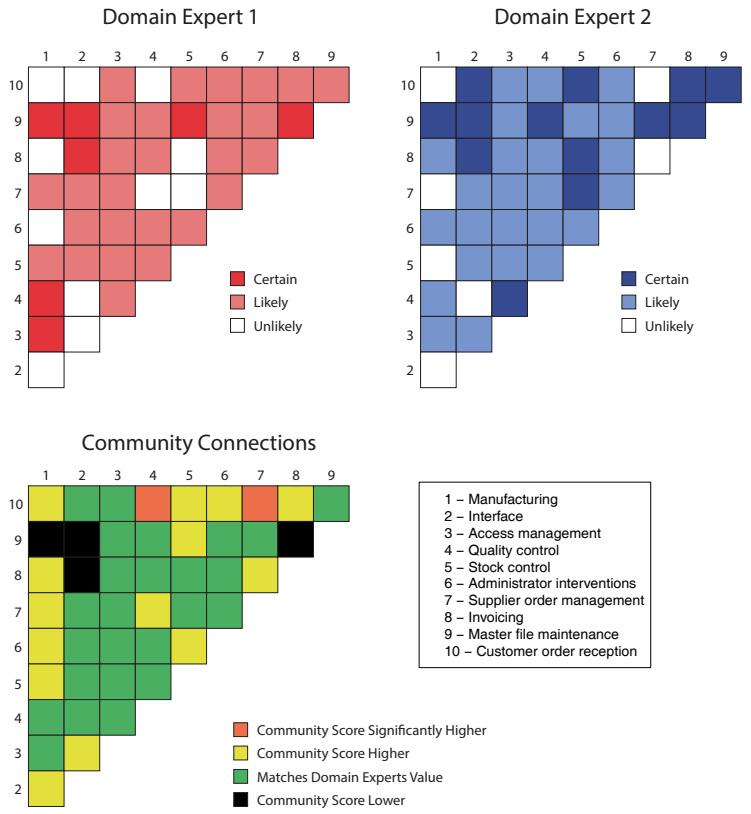

Figure 20: The answers given by domain experts and a comparison of their average with community-based assessment. (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

While the information possessed by domain experts is invaluable in the adoption process it is visible that there is a difference in their opinions which is significant in some cases like the connections of Stock Control and Invoicing or Stock Control and Supplier order management. Since the domain experts have no ready knowledge on every single program these differences are inevitable. While this assessment only involved the highest level of features which are very general we can imagine how much harder this task could be on a level with tens or even hundreds of features.

Furthermore, it is visible that the green fields are in majority which can mean that the community detection output is similar to the average of domain expert views, thus holds real and potentially useful information about the dependencies

between features. We note that the average of the votes of domain experts in the figure is based on the rounded down value of average, in case we would round up we get more black fields but also even more green fields. This shows that if domain experts take a more cautious approach the data of the communities can match even better with their views. Our results also fall closer to the answers of the domain experts individually than they do to each other, the absolute difference between them being 24 while the community dependency shows 20 and 22 absolute difference from them.

Although we can see that the communities can contain similar information to what the domain experts use we also know that they build on structural information as they are working on the call graph of the system. This means that it contains additional information that highlights a more structural level and while calls inside the system do not occur necessarily in every case, it is still worth to take them into account even for a quick inspection.

Answer to RQ2: According to the evaluation the call graph communities represent real feature coupling information that approximates domain expert knowledge well while relying only on structural information, thus can contribute to the decisions of a domain expert by providing another informed viewpoint.

6.3. Discussion

In this subsection we overview the possible uses of our methods presented. Figure 21 highlights how various feature extraction techniques can be used to help in building the new product line architecture.

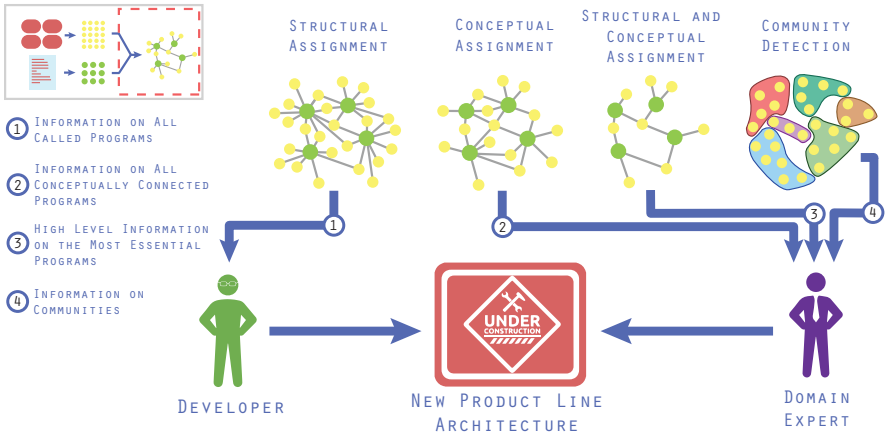

Figure 21: The possible ways of usage of the results of various feature extraction techniques in helping product line adoption

- Structural Extraction - Provides a detailed, widespread analysis. It is good for developers since they are required to have knowledge of all of the programs called by a feature.

- Conceptual Extraction - For domain experts, on the other hand, all called programs can be too much. This approach introduces conceptual dependencies but may contain too much noise for smooth work.
- Combination (ESS) - Grasps the essence of features, more fit for domain experts. While constructing the new architecture, the domain experts need to judge properly which parts of the variants should be adopted. In this decision making process, the results of this combined extraction highly decrease complexity and it can also facilitate test planning in the future.
- Community detection - Identifies program communities over the call graph that work closely together while having less outer relations.
- Community matching - Matches program communities with features in the code. Can be used to find differences between the view of the domain experts and the actual implementation of features.

As our evaluation pointed out, the developers found the programs of our result sets as a source of feature mapping information much more preferable to the other, unrelated programs.

Besides these, we would like to mention some other possible ways to use the results. Firstly, connections are not necessarily observable through the calls of the system, programs can, for instance, connect by accessing to the same data objects. This means that not every connection will present itself on the call graph. These, however, can be found via conceptual feature extraction, since it is likely that programs using the same data are conceptually connected to the same feature. This is why the programs detected by the conceptual extraction and not discovered via structural information can still be valuable. This seems to be verified by the developers also since in our evaluation we found that only information retrieval based results scored even higher in their regard than the call graph based results. However noisy, conceptual data still represents a ready source of the semantically connected programs to each feature, hence it can also be a useful information source.

Additionally, both the structural and information retrieval based methods can be tailored according to our intent by filtering out the more general programs of the call graph which provides the possibility to form even better separated program sets and by changing semantic similarity thresholds in the conceptual method.

As our evaluation pointed out communities hold valid information about features and similar opinions can be extracted from it as relying on domain expertise while working on a more structural level. Although community detection and matching can aid the understanding of features and the current state of the system they can also be useful in the future at a maintenance stage. Additional possible uses even involve the refinement of the feature model and easily comprehensible tracking of the evolution of programs. Some appropriate setting of

communities could also be optimized for a recommendation system of possible splitting or merging of features.

We made our results available to our industrial partner, who has already commenced on constructing the new SPL architecture. The work is proceeding well, product line adoption seems to go according to the plans and the results of our experiments are utilized in the process.

Our current project required us to remain within the field of 4GL languages, but our future plans involve also implementing our approach in more traditional languages like Java or C++. Since call graphs can be constructed for these languages too and they also tend to contain a lot of natural language text, our method can be adapted to work in more traditional conditions. Magic and 4GL languages, in general, are generally more data intensive, and in our specific advantageous case features were readily connected with menu elements making direct calls inside the software, but a similar menu to class position can also be rather easily achieved in a lot of Java or C++ applications due to the confinements of some frameworks.

7. 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, although only a few works are published about the automatic analysis and modeling of 4GL or specifically Magic applications. The maintenance of Magic applications is supported by cost estimation and quality analysis methods [24, 25, 17]. Architectural analysis, reverse engineering and optimization are visible topics in the Magic community [15, 26, 9, 8], and after some years of Magic development migration to object-oriented languages [27] as well.

SPL has a widespread literature, and over the last 8-10 years it has gained even more popularity. All three phases of feature analysis (identification, analysis, and transformation) are tackled by researchers. A recommended mapping study on recent works on feature location can be read in [5].

In the recent years efficient community detection algorithms have been developed which can cope with very large graphs with millions of nodes and potentially billions of edges [28]. Originally, community detection was applied mostly on graphs that represent complex networks (e.g. social, biological, technological) [29], and have also been suggested for software engineering problems [20]. Besides applications in testing and dependency analysis, software modularization [21] is also addressed by community algorithms. Although modularization is related to reuse and a natural extension of the approach is to use them for feature analysis purposes, we are not aware of previous work on features and community algorithms in 4GL context. Graph-based methods in general usually yield good results and scale well to large systems. For example, in [30] Xue et. al. introduce a model differencing technique to detect evolutionary changes to

product features. In this process, they only rely on feature sets, while we pointed out that the relationships with communities can also be a valuable information.

Software product line extraction is a time-consuming task. To speed up this activity, many semi-automatic approaches has been proposed [31, 32, 33]. Reverse engineering is a popular approach which has recently received an increased attention from the research community. With this technique missing parts can be recovered, feature models can be extracted a set of features, etc. [31, 34]. Applying these approaches companies can migrate their system into a software product line. However, changing to a new development process is risky and may have unnecessary costs. The work of Krüger et al. [35] supports cost estimations for the extractive approaches and provides a basis for further research.

Feature models are considered first class artifacts in variability modeling. Haslinger et al. [33] 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. [36] analyze Linux kernel (which is a standard subject in variability analysis) configurations to obtain feature models. LSI has been applied for recovering traceability links between various software artifacts, even in feature extraction experiments [37, 38, 39]. The work of Marcus and Maletic [18] is an early paper on applying LSI for this purpose. Eyal-Salman et al. [6, 39] use LSI for recovering traceability links 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. The main contrast between Eyal-Salman et al.[39] and our methods is that instead of using family models and test cases to refine LSI results, we used the information of call graphs and communities for this purpose. IR-based solution for feature extraction is combined with structural information in the work of Al-msie'deen et al. [11]. Further research deals with constraints in a semi-automatic way [40] both for functional and even nonfunctional [41] feature requirements. For an overview of analysis methods in product line research, we refer the interested reader to the survey of Thum et al. [42].

Substantial research on similar fields already pointed out that features are not independent of each other, and the structure of source code resembles the structure of features [30, 43, 44]. Building on this intuition, metrics can be defined [44] based on structural similarity and can be used in a feature location method within an iterative context-aware approach. Besides these metrics, communities also provide relevant information about the structure of program code. In further contrast, our approach takes into account the internal structure of software for determining the relevance of the program elements to the features.

Several studies [45, 46, 47, 48] have shown, that variability analysis across different software levels is able to come up with promising results. For example, in [47] the authors presented an approach for commonality and variability analysis across multiple software projects at different levels of granularity. They evaluated their work on 19 C/C++ operating systems, while our approach was tested in Magic programming language. Feature levels are also a novelty in variability analysis.

Pairwise and t-wise testing are leading methods to product line testing to

address variance of features [49, 50], where similarity methods are employed as well [51, 52]. A possible future goal could be to make the system dynamically configurable, which is a problem known as Dynamic SPL [53, 54, 55, 56, 57, 58, 59, 60]. Today, many application domains demand runtime reconfiguration, for which the two main limitations are handling structural changes dynamically and checking the consistency of the evolved structural variability model during runtime [61]. However, as reported in [62], the current research on runtime variability is still heavily based on the decisions made during design time. Coping with uncertainty of dynamic changes also needs to be addressed [63].

Static methods individually are commonly used for the detection of features [43, 4, 48], but the combination of those is not a common practice in the field. Hybrid approaches [38] combine two or more types of analysis with the goal of using one type of analysis to compensate for the limitations of another, thus achieving better results. Dynamic analysis[43, 64, 65] is utilized in some of the related research to analyze execution scenarios by using methods which collect and analyze information about the execution of the artifacts. Search-based strategies [66, 45] applies algorithms from optimization field, like genetic algorithms in the localization features.

As we can see, product line adoption is a very widespread area, and the evaluation of the research results is often a challenging task. In a traditional programming language environment several approaches were introduced to overcome these obstacles. For example in a similar work [67] where structural and lexical information were combined, researchers used the well-known precision and recall metrics to compare their approach against state-of-the-art techniques. This method is similarly used in several works [68, 69, 37, 70]. Although the evaluation process is qualitative and seems appropriate, it is not applicable in every case [71, 72, 73] which is even truer for non-traditional programming languages like Magic. Applying existing approaches in 4GL encounters several obstacles, since the entire environment is very different. First of all there is no source code in its traditional sense, and these techniques usually rely heavily on it. The structure of these systems also differs from the conventional projects. Finding an open source project is a very hard task. It is clear that the evaluation in these cases should be unique, and new approaches are needed.

Several existing approaches can be adapted to the 4GL environment, although none of the papers we encountered cope with 4GL product line adoption directly.

8. Conclusions

An ongoing industrial project was presented, which undergoes a software product line adoption process. In this work, we concentrated on the feature extraction and analysis aspects of the project, which are fundamental parts of the effort because further architecture redesign and implementation are and will be based on this information. For feature extraction we used two approaches: one based on computing structural information in form of call-graphs, and the other extracted from the textual representation of high level feature models and

the code. The combination of the two pieces of information had to be performed and processed in such a way that the resulting models are most useful for project participants. Experimental results show that the final models are significantly more comprehensible, and hence directly usable (though in various forms) by domain experts, architects and developers. On the other hand, the high level view of domain experts is not necessarily in line with the actual implementation. We introduced community detection methods on the call structure of the system to match communities with feature code originated from domain experts. We evaluated our results based on comparison with human expertise and shown that our various outputs present knowledge that matches well with the thinking of experts but can still highlight additional insights. The proposed method including the associated toolset is currently in use by our industrial partner in this ongoing effort. Although the approach was implemented in Magic, a 4GL technology, we believe that the fundamental method could be suitable for other more traditional paradigms as well after the necessary adaptations.

Acknowledgment

Ferenc Kocsis was supported in part by the Hungarian National Grant GINOP-2.1.1-15-2015-00370. András Kicsi, Viktor Csuvik, László Vidács, Ferenc Horváth and Tibor Gyimóthy were supported in part by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). Árpád Beszédes was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. The Ministry of Human Capacities, Hungary grant 20391-3/2018/FEKUSTRAT is also acknowledged.

References

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

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

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

[4] C. Kästner, A. Dreiling, K. Ostermann, Variability Mining: Consistent Semi-automatic Detection of Product-Line Features, IEEE Transactions on Software Engineering 40 (1) (2014) 67–82.

[5] W. K. G. Assunção, 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, ACM Press, New York, New York, USA, 2014, pp. 52–59.

[6] H. Eyal-Salman, A.-D. Seriai, C. Dony, 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, ACM Press, New York, New York, USA, 2012, pp. 21–25.

[7] Magic Software Enterprises Ltd., Magic Software Enterprises, http://www.magicsoftware.com (last visited May 2017).

[8] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, 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, 2010, pp. 1–6.

[9] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, 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, 2011, pp. 343–346.

[10] A. Kicsi, L. Vidács, A. Beszédes, F. Kocsis, I. Kovács, Information retrieval based feature analysis for product line adoption in 4gl systems, in: Proceedins of the 17th International Conference on Computational Science and Its Applications – ICCSA 2017, IEEE, 2017, pp. 1–6.

[11] R. Al-msie'deen, A.-D. Seriai, M. Huchard, C. Urtado, 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, 2013, pp. 586–593.

[12] P. C. Clements, L. G. Jones, J. D. McGregor, L. M. Northrop, Getting there from here: a roadmap for software product line adoption, Communications of the ACM 49 (12) (2006) 33.

[13] P. Clements, C. Krueger, Eliminating the adoption barrier, IEEE Software 19 (2002) 29–31.

[14] C. Catal, Cagatay, Barriers to the adoption of software product line engineering, ACM SIGSOFT Software Engineering Notes 34 (6) (2009) 1.

[15] J. V. Harrison, 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.

[16] M. Ballarin, R. Lapeña, C. Cetina, Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products, in: Proceedings of the 15th International Conference on Software Reuse:

Bridging with Social-Awareness - Volume 9679, Springer-Verlag New York, Inc., 2016, pp. 215–230.

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

[18] A. Marcus, 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.

[19] D. Falessi, G. Cantone, 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, ACM Press, New York, New York, USA, 2010, p. 1.

[20] J. Hamilton, S. Danicic, Dependence communities in source code, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, 2012, pp. 579–582.

[21] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Transactions on Software Engineering 32 (3) (2006) 193–208.

[22] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, R. A. Harshman, Indexing by Latent Semantic Analysis, Journal of the American Society of Information Science 41 (6) (1990) 391–407.

[23] A. Kicsi, V. Csuvik, L. Vidács, Á. Beszédes, T. Gyimóthy, Feature level complexity and coupling analysis in 4GL systems, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 10964 LNCS, Springer, Cham, 2018, pp. 438–453.

[24] J. Verner, G. Tate, Estimating Size and Effort in Fourth-Generation Development, IEEE Software 5 (1988) 15–22.

[25] G. Witting, G. Finnie, Using Artificial Neural Networks and Function Points to Estimate 4GL Software Development Effort, Australasian Journal of Information Systems 1 (2) (1994) 87–94.

[26] Ocean Software Solutions, Homepage of Magic Optimizer, http://www.magic-optimizer.com (last visited May 2017).

[27] M2J Software LLC, Homepage of M2J, http://www.magic2java.com (last visited May 2017).

[28] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, Journal of statistical mechanics: theory and experiment 2008 (10) (2008) P1000.

[29] S. Fortunato, Community detection in graphs, Physics reports 486 (3) (2010) 75–174.

[30] Y. Xue, Z. Xing, S. Jarzabek, Understanding feature evolution in a family of product variants, Proceedings - Working Conference on Reverse Engineering, WCRE (2010) 109–118.

[31] M. T. Valente, V. Borges, L. Passos, A Semi-Automatic Approach for Extracting Software Product Lines, IEEE Transactions on Software Engineering 38 (4) (2012) 737–754.

[32] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, A. Egyed, Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants, Empirical Software Engineering 22 (4) (2017) 1763–1794.

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

[34] C. Lima, C. Chavez, E. S. de Almeida, Investigating the Recovery of Product Line Architectures: An Approach Proposal, Springer, Cham, 2017, pp. 201–207.

[35] J. Krüger, W. Fenske, J. Meinicke, T. Leich, G. Saake, Extracting software product lines: a cost estimation perspective, in: Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16, ACM Press, New York, New York, USA, 2016, pp. 354–361.

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

[37] Y. Xue, Z. Xing, S. Jarzabek, Feature location in a collection of product variants, Proceedings - Working Conference on Reverse Engineering, WCRE (2012) 145–154.

[38] R. Al-msie, a. D. Seriai, M. Huchard, C. Urtado, An approach to recover feature models from object-oriented source code, Tech. rep. (2012).

[39] H. Eyal-Salman, A. D. Seriai, C. Dony, Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval, Proceedings of the 2013 IEEE 14th International Conference on Information Reuse and Integration, IEEE IRI 2013 (2013) 209–216.

[40] E. Bagheri, F. Ensan, D. Gasevic, Decision support for the software product line domain engineering lifecycle, Automated Software Engineering 19 (3) (2012) 335–377.

[41] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, G. Saake, SPL Conqueror: Toward optimization of non-functional properties in software product lines, Software Quality Journal 20 (3-4) (2012) 487–517.

[42] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A Classification and Survey of Analysis Strategies for Software Product Lines, ACM Computing Surveys 47 (1) (2014) 1–45.

[43] B. Klatt, M. Küster, A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies, Tech. rep. (2013).

[44] X. Peng, Z. Xing, X. Tan, Y. Yu, W. Zhao, The Journal of Systems and Software Improving feature location using structural similarity and iterative graph mapping, The Journal of Systems & Software 86 (2013) 664–676.

[45] M. I. Ullah, G. Ruhe, V. Garousi, Decision support for moving from a single product to a product portfolio in evolving software systems, The Journal of Systems & Software 83 (2010) 2496–2512.

[46] K. Valincius, V. Stuikys, R. Damasevicius, Understanding of e-commerce is through feature models and their metrics, Proceedings of the IADIS International Conference Information Systems 2013, IS 2013 8 (1) (2013) 55–62.

[47] M. B. Kelly, J. S. Alexander, B. Adams, A. E. Hassan, Recovering a Balanced Overview of Topics in a Software Domain (2011).

[48] P. Paškevičius, R. Damaševičius, E. Karčiauskas, R. Marcinkevičius, Automatic extraction of features and generation of feature models from java programs, Information Technology and Control 41 (4) (2012) 376–384.

[49] G. Perrouin, S. Sen, J. Klein, B. Baudry, Y. le Traon, Automatic and Scalable T-wise Test Case Generation Strategies for Software Product Lines, Proc. of Intl. Conf. on Software Testing 215483 (2010).

[50] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, G. Saake, IncLing: efficient product-line testing using incremental pairwise sampling, in: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences - GPCE 2016, Vol. 52, ACM Press, New York, New York, USA, 2016, pp. 144–155.

[51] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, Y. Le Traon, Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines, IEEE Transactions on Software Engineering 40 (7) (2014) 650–670.

[52] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, G. Saake, Similarity-based prioritization in software product-line testing, in: Proceedings of the 18th International Software Product Line Conference on - SPLC '14, ACM Press, New York, New York, USA, 2014, pp. 197–206.

[53] K. Lee, K. C. Kang, J. Lee, Concepts and Guidelines of Feature Modeling for Product Line Software Engineering, in: Software Reuse Methods Techniques and Tools, Vol. 2319, Springer, Berlin, Heidelberg, 2002, pp. 62–77.

[54] L. Baresi, C. Quinton, Dynamically Evolving the Structural Variability of Dynamic Software Product Lines, 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (2015).

[55] J.-M. Horcas, M. Pinto, L. Fuentes, Runtime Enforcement of Dynamic Security Policies, Springer, Cham, 2014, pp. 340–356.

[56] N. Gamez, L. Fuentes, J. M. Troya, Creating Self-Adapting Mobile Systems with Dynamic Software Product Lines, IEEE Software 32 (2) (2015) 105–112.

[57] M. Bashari, E. Bagheri, W. Du, Dynamic Software Product Line Engineering: A Reference Framework, International Journal of Software Engineering and Knowledge Engineering 27 (02) (2017) 191–234.

[58] A. G. Uchôa, C. I. M. Bezerra, I. C. Machado, J. M. Monteiro, R. M. C. Andrade, ReMINDER: An Approach to Modeling Non-Functional Properties in Dynamic Software Product Lines, Springer, Cham, 2017, pp. 65–73.

[59] M. Hinchey, S. Park, K. Schmid, Building Dynamic Software Product Lines, IEEE Computer Society 45 (10) (2012) 22–26.

[60] J. Lee, A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering, 10th International Software Product Line Conference (2006) 131–140.

[61] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, M. Hinchey, An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry, Journal of Systems and Software 91 (1) (2014) 3–23.

[62] N. Bencomo, J. Lee, S. Hallsteinsen, How dynamic is your Dynamic Software Product Line?, DiVA project (EU FP7 STREP) (2010) 61–67.

[63] A. Classen, A. Hubaux, F. Sanen, E. Truyen, J. Vallejos, P. Costanza, W. De Meuter, P. Heymans, W. Joosen, Modelling Variability in Self-Adaptive Systems: Towards a Research Agenda, Proceedings of International Workshop on Modularization, Composition and Generative Techniques for Product-Line Engineering 1 (2) (2008) 19–26.

[64] A. Olszak, B. N. Jørgensen, Remodularizing Java programs for comprehension of features, Proceedings of the First International Workshop on FeatureOriented Software Development FOSD 09 (2009) 19–26.

[65] M. A. Maia, V. Sobreira, K. Paixão, S. A. Amo, I. R. Silva, Using a sequence alignment algorithm to identify commonalities and variabilities from execution traces, Tech. rep. (2008).

[66] S. Lohar, S. Amornborvornwong, A. Zisman, J. Cleland-Huang, Improving trace accuracy through data-driven configuration and composition of tracing features, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013, ACM Press, New York, New York, USA, 2013, p. 378.

[67] E. Hill, L. Pollock, K. Vijay-Shanker, Exploring the neighborhood with dora to expedite software maintenance, in: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07, ACM Press, New York, New York, USA, 2007, p. 14.

[68] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, Fuqing Yang, SNIAFL: towards a static non-interactive approach to feature location, in: Proceedings. 26th International Conference on Software Engineering, IEEE Comput. Soc, 2004, pp. 293–303. doi:10.1109/ICSE.2004.1317452.

[69] R. Al-Msie'Deen, A. D. Seriai, M. Huchard, C. Urtado, S. Vauttier, Mining features from the object-oriented source code of software variants by combining lexical and structural similarity, in: Proceedings of the 2013 IEEE 14th International Conference on Information Reuse and Integration, IEEE IRI 2013, no. January, IEEE, 2013, pp. 586–593.

[70] A. Marcus, A. Sergeyev, V. Rajlieh, J. I. Maletic, An information retrieval approach to concept location in source code, in: Proceedings - Working Conference on Reverse Engineering, WCRE, IEEE Comput. Soc, 2004, pp. 214–223.

[71] R. Al-Msie'Deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, H. E. Salman, Feature location in a collection of software product variants using formal concept analysis, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7925 LNCS, Springer, Berlin, Heidelberg, 2013, pp. 302–307.

[72] D. Poshyvanyk, Y. G. Guéhéneuc, A. Marcus, G. Antoniol, V. Rajlich, Combining probabilistic ranking and latent semantic indexing for feature identification, in: IEEE International Conference on Program Comprehension, IEEE, 2006, pp. 137–146.

[73] D. Poshyvanyk, A. Marcus, Combining formal concept analysis with information retrieval for concept location in source code, in: IEEE International Conference on Program Comprehension, IEEE, 2007, pp. 37–46. doi:10.1017/S0024282913000583.

Appendix A. Matching of communities with 1st level features

Communities and matching high level features are shown in Figure A.22

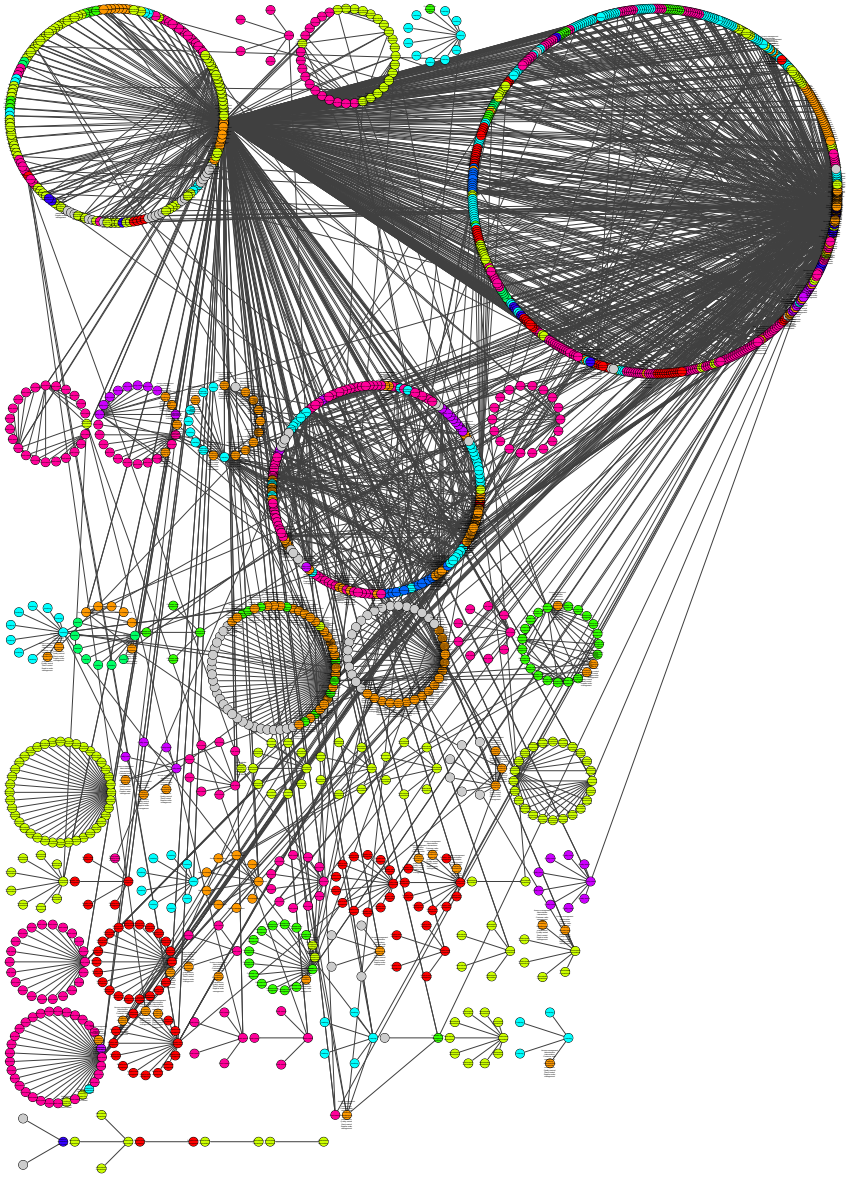

Figure A.22: Matching communities with high level features. Each node represents a program of the system, while its color marks its feature affiliation. See the node text for cases of more than one features assigned. The edges are the calls the programs make while the circular groups represent their assigned communities. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Appendix B. Matching of communities with 2nd level features

Communities and matching second level features are shown in Figure B.23

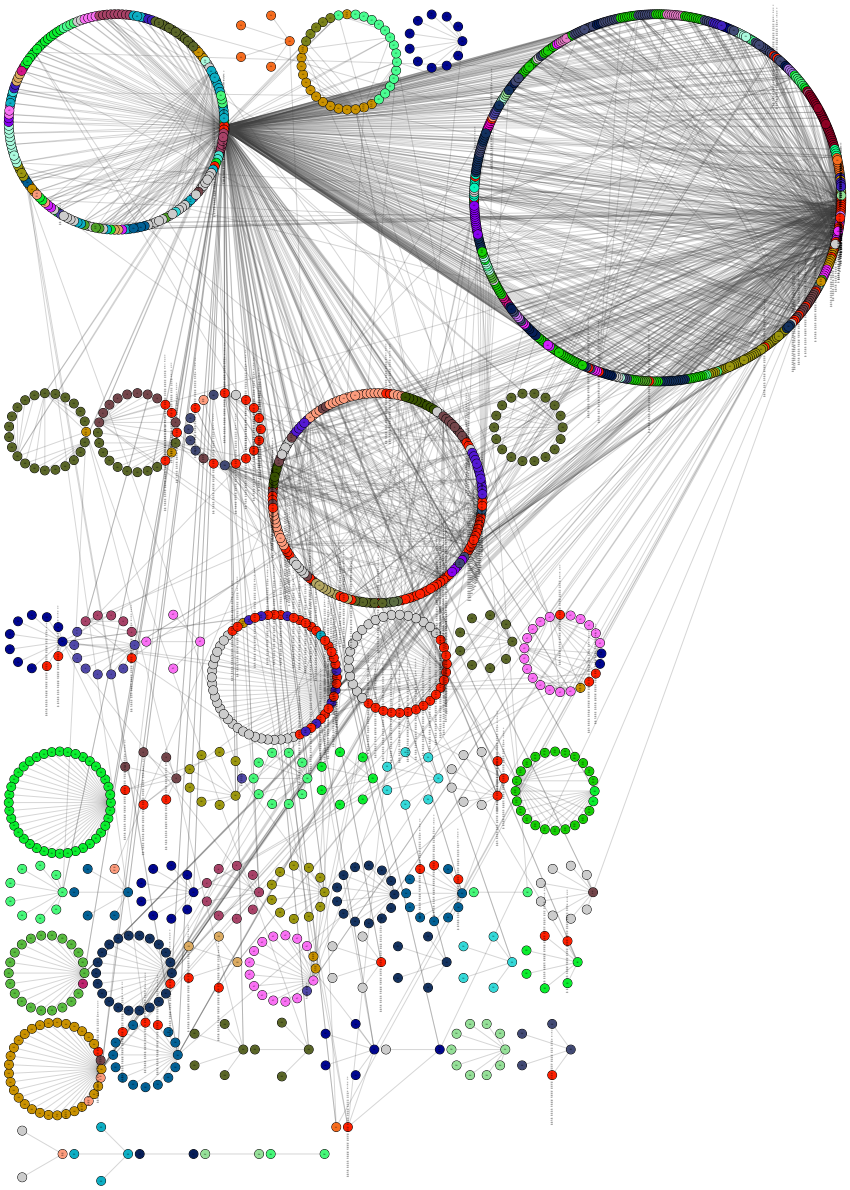

Figure B.23: Matching communities with second level features. Each node represents a program of the system, while its color marks its feature affiliation. See the node text for cases of more than one features assigned. The edges are the calls the programs make while the circular groups represent their assigned communities. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)