Towards Clone-And-Own Support: Locating Relevant Methods in Legacy Products*

Raúl Lapeña
SVIT Research Group
Universidad San Jorge
Zaragoza, Spain
rlapena@usj.es

Manuel Ballarin
SVIT Research Group
Universidad San Jorge
Zaragoza, Spain
mballarin@usj.es

Carlos Cetina
SVIT Research Group
Universidad San Jorge
Zaragoza, Spain
ccetina@usj.es

ABSTRACT

Clone-and-Own (CAO) is a common practice in families of software products consisting of reusing code from methods in legacy products in new developments. In industrial scenarios, CAO consumes high amounts of time and effort without guaranteeing good results. We propose a novel approach, Computer Assisted CAO (CACAO), that given the natural language requirements of a new product, and the legacy products from that family, ranks the legacy methods in the family for each of the new product requirements according to their relevancy to the new development. We evaluated our approach in the industrial domain of train control software. Without CACAO, software engineers tasked with the development of a new product had to manually review a total of 2200 methods in the family. Results show that CACAO can reduce the number of methods to be reviewed, and guide software engineers towards the identification of relevant legacy methods to be reused in the new product.

CCS Concepts

•**Software and its engineering** → **Reusability;**

Keywords

Clone and Own; Software Reuse; Families of Software Products

1. INTRODUCTION

Clone-and-Own (CAO) [2, 5, 18, 21, 23] is a common practice in the development of new products in families of software products. It consists of reusing code from legacy products, modifying it to comply with the functionality particularities of the new product. Code reuse enables faster software development and easier tracking of projects, and helps maintain the code style consistent between products.

In the practice, CAO is carried out manually and relies on the knowledge that software developers have of the family. In industrial scenarios, families of software products tend to have a myriad of products with long and complex implementations, coded and maintained over long periods of time by different developers. In these scenarios, engineers tasked with new product developments often lack knowledge over the entirety of the products and their implementation details. Under these conditions, CAO is a process that consumes high amounts of time and effort without guaranteeing good results.

In this paper, we propose a novel approach, named Computer Assisted CAO (CACAO), that leverages Part-of-Speech tagging (POS tagging) [9] and adapts Latent Semantic Indexing (LSI) [12] to rank the relevancy of legacy products for a new development at the requirements level, and to locate their most significant methods for each of the new product requirements.

Given the natural language specifications of a new product in a family of software products, and the legacy products that belong to it, our approach detects which are the legacy products that are the closest to the new product in terms of requirements. In a second step, our approach searches the code of the closest legacy products for methods that are relevant for the new product requirements. As a result, our approach produces a code relevancy ranking for each of the requirements of the new product. Software engineers can benefit from the rankings to avoid the mentioned CAO issues.

We evaluated our approach in the industrial domain of railway control software. Our industrial partner, Construcciones y Auxiliar de Ferrocarriles (CAF), provided a family of five software products used to control the trains they manufacture. In our evaluation, one product acts as a new product in the family, and the rest act as legacy products. The code of the product that acts as the new product is used as an oracle. We apply our approach to that scenario, and measure its performance in terms of recall and precision [27] by comparing the results to the code of the oracle. These steps are followed five times, having all the products play the role of the oracle.

Results show that it is likely to find relevant code in the rankings. With CACAO, the amount of methods that software engineers review when developing a requirement for a new product is reduced: it is only needed to review a percentage of the original 2200 methods to build a new product.

*This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the project Model-Driven Variability Extraction for Software Product Line Adoption (TIN2015-64397-R).

SPLC '16, September 16-23, 2016, Beijing, China

DOI: http://dx.doi.org/10.1145/2934466.2934485

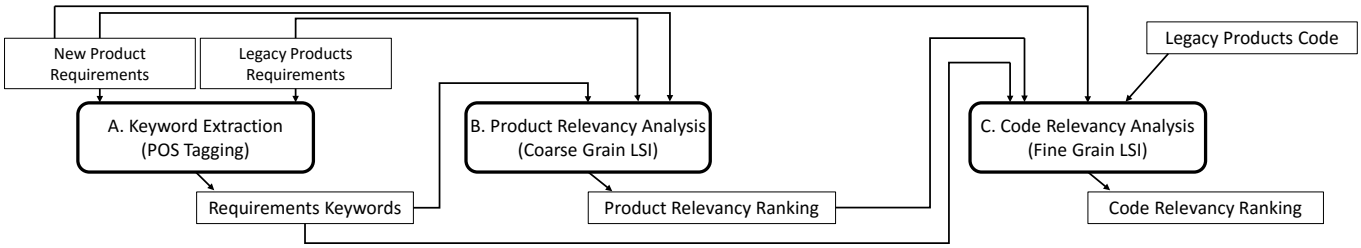

Figure 1: Approach Overview

The remainder of the paper is structured as follows: Section 2 presents our approach and shows how to apply it to a running example. Section 3 shows the evaluation of our work. Section 4 discusses the results of our work. Section 5 postulates the threats to the validity of our work. Section 6 comprehends the works related to this paper. Section 7 presents the conclusions of our work.

2. APPROACH

The goal of our approach is, given a set of natural language requirements for a new product, and the legacy products, to provide code relevancy rankings that enable software engineers to reduce the amount of methods they must review to develop the new product. To this extent, a series of steps are followed (see Figure 1):

A. First, the relevant keywords from the new product requirements and the legacy product requirements are extracted through extended POS Tagging techniques.

B. The second step of our approach performs a Coarse Grain LSI (CG-LSI) process to detect which of the legacy products are the closest to the new product in terms of requirements.

C. The third and last step of our approach is to perform a Fine Grain LSI (FG-LSI) process at the code level to detect which of the methods in the close legacy products are related to the new product requirements.

In the following pages, we detail the steps of our approach in the above order. To illustrate them, we use a running example from our industrial partner, CAF (Construcciones y Auxiliar de Ferrocarriles, at http://www.caf.net/en). CAF is a worldwide leader company in the railway industry. Since its foundation more than 100 years ago, they develop rail solutions such as high speed trains, regional and commuter trains, metros, trams and Light Rail Vehicles.

2.1 Keyword Extraction

The first step of our approach extracts keywords from the natural language requirements of the new product and the legacy products in the family. There are plenty of techniques that perform text mining and information retrieval from natural language requirements such as the ones in [8, 3, 1, 6]. The analysis of POS tags in search for nouns and nominal structures in documents has shown promising results when extracting keywords from technical documents [9, 15]. The removal of stopwords (frequently occurring words meaningless to information retrieval) also helps produce more accurate results when mining data in documents [25, 16, 29].

The combination of the analysis of POS tags and removal of stopwords is a frequent practice that our approach adopts to extract the most relevant keywords from the requirements documents. First, our approach searches for domain terms, provided by the software engineers, in the requirements. Then, the POS tags of the words that form the requirements, domain terms excluded, are analyzed. Afterwards, the words are filtered by their syntactic role in the sentences, and finally refined with a set of stopwords, also provided by the software engineers.

In Figure 2, a requirement from our running example is provided. This requirement describes part of the functionality of the pantograph of a train. The pantograph is the element that is used to harvest energy from the overhead wires installed in train lines.

First, the terms from the list of domain terms present in the requirement are subtracted from the requirement and introduced into the keywords list. Afterwards, the POS tags of the words that compose the requirement, domain terms excluded, are extracted. In Figure 2, the result of tagging the example requirement is shown. In the figure, it is possible to appreciate words like 'panto' or 'doors' as nouns, and 'inhibit' or 'close' as verbs. The rest of the words are omitted in the figure.

After the POS tagging, a filtering process takes place. Nouns are taken as keyword candidates due to their importance for keyword extraction [9]. The rest of the words are discarded. Nouns are filtered with a set of stopwords provided by the software engineers. The nouns that do not belong to the list of stopwords are added to the keywords list. Figure 2 shows a sample of the stopwords provided by a software engineer, as well as the final list of keywords extracted from the example requirement.

Keywords from all the requirements in all the documents are combined into a single set of terms, removing duplicates. The output of the first step of our approach is a set of terms with all the keywords extracted. These terms are used in the next steps of our approach.

2.2 Product Relevancy Analysis

In the second step of our approach, the keywords are used along with the new product and the legacy products requirements to perform a Coarse Grain LSI (CG-LSI). The aim of the CG-LSI process is to order the legacy products in a ranking that reflects their similarity to the new product development in terms of requirements.

Carrying out this step of our approach is relevant in industrial domains, where software families are conformed by a myriad of legacy products. In these scenarios, developers of a new product may lack knowledge of all the legacy

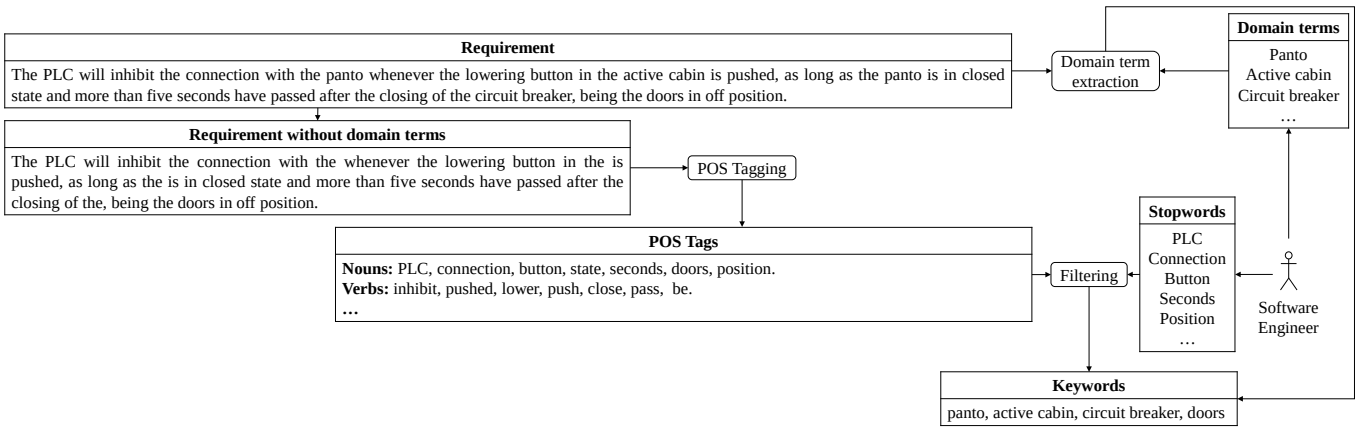

Figure 2: Keyword Extraction

products details. Through the product ranking, developers can appreciate whether the legacy products they know are relevant for the new development.

LSI [22] is an automatic mathematical/statistical technique that analyzes relationships between *queries* and *documents* (bodies of text). It constructs vector representations of both a user *query* and a corpus of text *documents* by encoding them as a *term-by-document co-occurrence matrix*, and analyzes the relationships between those vectors to get a similarity ranking between the *query* and the *documents*.

In the second step of our approach, we adapted LSI to extract a ranking of the legacy products according to their similarity to the new product in terms of requirements. In our adapted CG-LSI, *terms* are the keywords extracted in the first step of our approach, *documents* are the legacy products requirements documents, and the *query* column is formed by the new product requirements document. Values of *term* occurrences in both the legacy product requirements documents and the new product requirements document are counted, and used to build the *term-by-document co-occurrence matrix*. The *documents* and the *query* are then transformed into vectors, and the relationships between the legacy product requirements documents and the new product requirements document are analyzed to extract the legacy product relevancy ranking.

Figure 3 shows the *term-by-document co-occurrence matrix* with the values associated to our running example, the vectors, and the resulting ranking. In the following paragraphs, an overview of the elements of the matrix is provided.

- Each row in the matrix stands for each unique keyword (*term*) extracted in the first step of our approach. In Figure 3, it is possible to appreciate a set of representative keywords in the domain such as 'PANTO' or 'DOORS' as the *terms* of each row.

- Each column in the matrix stands for the requirements document of each legacy product. In Figure 3, it is possible to appreciate the names of the legacy products in the columns such as 'KAOHSIUNG' or 'AUCKLAND', representing the requirements *documents* of those products.

- The final column stands for the *query*. In our approach, the *query* column stands for the requirements of the new product. In Figure 3, the name of the new product in the *query* column ('CINCINNATI') represents its requirements *document*.

- Each cell in the matrix contains the frequency with which the *term* of its row appears in the *document* denoted by its column. For instance, in Figure 3, the *term* 'PANTO' appears 114 times in the 'AUCKLAND' legacy product and 150 times in the 'CINCINNATI' new development.

We obtain vector representations of the *documents* and the *query* by normalizing and decomposing the *term-by-document co-occurrence matrix* using a matrix factorization technique called *singular value decomposition* (SVD) [12]. SVD is a form of factor analysis, or more properly the mathematical generalization of which factor analysis is a special case. In SVD, a rectangular matrix is decomposed into the product of three other matrices. One component matrix describes the original row entities as vectors of derived orthogonal factor values, another describes the original column entities in the same way, and the third is a diagonal matrix containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed.

In Figure 3, a three-dimensional graph of the SVD is provided. On the graph, it is possible to appreciate each product, represented in the form a vector. The graph reflects the 'Houston' train vector as the closest to the new product vector, followed by the 'Budapest' train vector.

To measure the similarity degree between vectors, our approach calculates the cosine between the *query* vector and the *documents* vectors. Cosine values closer to one denote a higher degree of similarity, and cosine values closer to minus one denote a lower degree of similarity. Similarity increases as vectors point in the same general direction (as more *terms* are shared between *documents*). Having this measurement, our approach orders the legacy products according to their similarity degree to the new product in terms of requirements. The most similar legacy products are the ones that can be of the most relevance to the development process of the new product.

As the output of the second step of our approach, the product relevancy ranking (which can be seen in Figure 3)

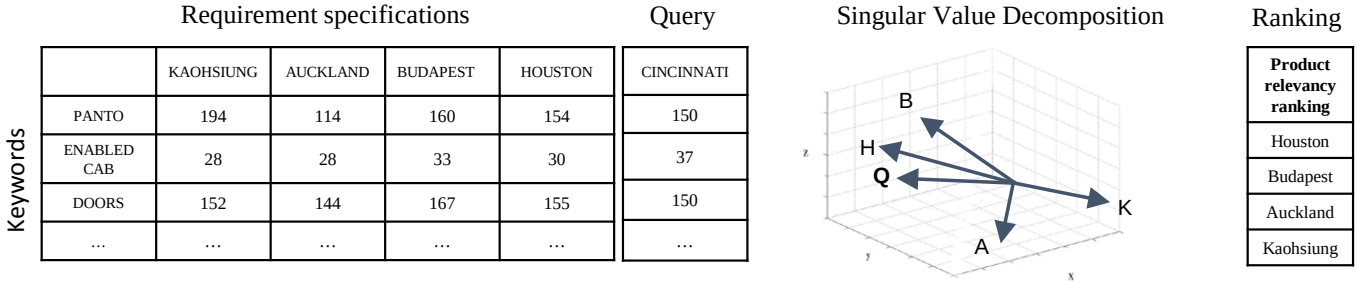

Figure 3: Product Relevancy Analysis

is produced according to the calculated similarity degrees. In our running example, our approach returns the legacy trains 'Houston' and 'Budapest' in the first and second position of the product relevancy ranking due to the cosines being '0.9243' and '0.8454', implying a high similarity degree with the new product in terms of requirements. On the opposite, the legacy train 'Kaohsiung' is returned in a latter position of the ranking due to its cosine being '-0.7836', a lower similarity degree.

The product relevancy ranking enables developers to decide whether to keep the legacy products familiar to them in the next step of CACAO, making a mixture between known and unknown products, or disregard them and only involve non-familiar products. Involving familiar products in the process is positive, since it is easier for software engineers to understand and reuse code known to them, but it should never enforce reusing code from non-relevant products.

2.3 Code Relevancy Analysis

In the third step of our approach, keywords are used along with the new product requirements, the product relevancy ranking, and the legacy products code to perform a Fine Grain LSI (FG-LSI) at the code level. The aim of the FG-LSI process is to order the methods of the legacy products in a ranking that reflects how similar they are to each of the new product requirements.

In the third step of our approach, we adapted LSI to extract a ranking of the methods in the relevant products that are of importance to the development of each new product requirement. In our adapted FG-LSI, *terms* are the keywords extracted in the first step of our approach, *documents* are the methods of the relevant legacy products, and there are several *query* columns, each of them a requirement of the new product development. Notice that in the third step of our approach, several instances of the *term-by-document co-occurrence matrix* are generated (one per query column). Values of *term* occurrences in both the methods and each of the requirements are counted, and used to build the matrices. The *documents* and the *queries* are transformed into vectors, and the relationships between the documents and each query are analyzed to extract a code relevancy ranking for each new product requirement.

For the sake of legibility, Figure 4 shows the LSI *term-by-document co-occurrence matrix* in a unified fashion (showing the values of the occurrences of the terms only once and grouping the queries to the right of the matrix). The figure also shows the values associated to this step of our running example and the resulting rankings. In the following paragraphs, an overview of the elements that a matrix contains is provided.

- Each row in the matrix stands for each of the unique keywords (*term*) extracted in the first step of our approach. In Figure 4, it is possible to appreciate a set of representative keywords in the domain such as 'PANTO' or 'DOORS' as the *terms* of each row.

- Each column in the matrix stands for each of the methods of the most relevant legacy products obtained in the previous step of our approach. A method *document* is composed by the name of the method, its variables, and the comments that appear in its body. External comments are not taken in account since we cannot ensure their belonging to a certain method. In Figure 4, columns M1 to MN represent the documents of those methods. Columns are labeled with method names, such as 'Detector_versions' or 'Propulsion_get_TCU'.

- In this step of our approach, there are several *query* columns. Each *query* column stands for each requirement of the new product. In Figure 4, columns R1 to RN represent the requirements of the new 'CINCINNATI' train. The top part of Fig. 2 shows the R1 requirement of the 'CINCINNATI' train.

- Each cell in the matrix contains the frequency with which the keyword of its row appears in the *document* denoted by its column. For instance, in Figure 4, the *term* 'PANTO' appears twice in both M1 and R1.

We use the SVD technique presented in the second step of our approach to calculate the vectors of the *documents* and the *query* for each one of the matrices. The vectors are represented in graphs similar to the one in Figure 3, that, for space reasons, were omitted in the figure.

For each graph, our approach calculates the cosines between the *query* vector and the *document* vectors to measure the similarity degrees between them. Having the measurement of the similarity of the legacy product methods with each requirement, and reasoning that the most similar legacy products methods to a particular requirement are the ones that can be of the most relevance to its development, we provide an ordered list of the legacy products methods for each requirement according to their relevance in the new requirement development.

As the output of the third step of our approach, the code relevancy rankings for each requirement (which can be seen in Figure 4) are returned. In our running example, our approach returns the legacy methods 'm1' and 'm2' in the

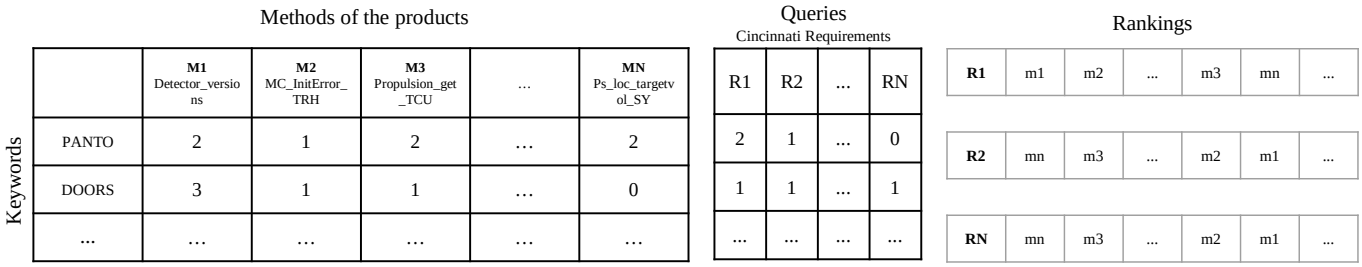

Figure 4: Code Relevancy Analysis

first and second position of the code relevancy ranking for the requirement 'R1' due to their cosines being '0.8743' and '0.6354', implying a high similarity degree between the code of those methods and the requirement. On the opposite, the legacy method 'mn' is returned in a latter position of the code relevancy ranking for the requirement 'R1' due to its cosine being '-0.7891', a lower similarity degree between code and requirement. This process is applied to all the requirements.

Software engineers in the company faced with the development of the new product can use these rankings to browse the most relevant methods for each requirement that they need to implement, avoiding the CAO issues.

3. EVALUATION

This section evaluates our approach by applying it to a case study from our industrial partner, comprising a product family composed by five trains with an average of about 420 requirements each. Requirements have an average of around 50 words. Trains are coded by an average of 550 methods, with an approximate extension of 310 LOC each. Therefore, each train is coded in about 170.5 KLOC, and the family comprehends about 2750 methods that account for around 852.5 KLOC.

3.1 Evaluation Steps

Figure 5 shows the steps followed to evaluate our approach. We use the products in the roles of either legacy or new products to perform CACAO and get method rankings. Methods of the legacy products in the rankings are then compared with the real code of the new product, which acts as an oracle, to obtain precision and recall values that enable further analysis of the method rankings.

First, roles are assigned to products in the family. One product acts as the new product and the rest act as legacy products. The requirements and code of the products that act as legacy products, and the requirements of the product that acts as the new product are used to perform CACAO, while the code of the latter is kept apart to be used as an oracle.

CACAO performs the steps described in our approach (see Section 2) to provide a method ranking for each requirement of the new product. Notice the dimensions of the rankings extracted by CACAO. Products, on average, feature 420 requirements and 550 methods. For a new product, on average, 420 rankings are generated. Taking in account all the legacy products in our set as relevant products for the new development, each ranking orders about 2200 methods on average.

Then, the methods that compose the rankings are compared one by one with the code of the oracle. We perform the code comparison by carrying out a diff not only because version control software is really popular, and therefore there is a wide amount of tool support that calculates differences between two source codes available, but also because code comparison techniques have been used successfully for large scale systems [13, 10], proving the computational cost of the operation to be affordable for large documents like ours.

The effectiveness of information retrieval techniques is typically measured by recall and precision [27]. For a given query, recall is defined as the percentage of retrieved documents that are relevant to the total number of relevant documents, and precision is defined as the percentage of retrieved documents that are relevant to the total number of retrieved documents. All measures have values between 0 and 1 [26]. We calculate the recall and precision for every method by analyzing the results of the diff.

In our evaluation, the recall of a certain method represents the percentage of the oracle that is covered by the method. The recall of a method is calculated by counting the number of equal code lines between the method and the oracle code and measuring it against the total number of code lines of the oracle. The formula that represents the recall of a method is as follows:

$$Recall(Method) = \frac{LOC(Method \cap Oracle)}{LOC(Oracle)}$$

The precision of a certain method, on the other hand, represents the percentage of the method that appears inside the code of the oracle. The precision of a method is calculated by counting the number of equal code lines between the method and the oracle code and measuring it against the total number of code lines of the method. The formula that represents the precision of a method is as follows:

$$Precision(Method) = \frac{LOC(Method \cap Oracle)}{LOC(Method)}$$

In both formulas, the LOC function retrieves the number of lines of the element contained inside the parentheses, and the intersection between the method and the oracle represents the lines of code that are common to both the method and the oracle.

The steps of the evaluation described in the previous paragraphs are repeated as many times as the number of products in the product family, changing the product that acts as the new product in every iteration until every product in the product set has acted as the new product.

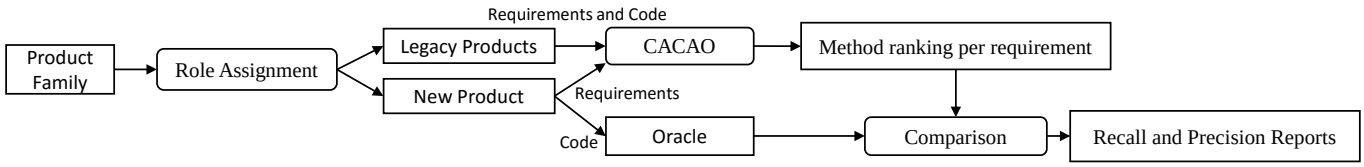

Figure 5: Evaluation Overview

3.2 Implementation Details

The different steps carried out to perform and evaluate CACAO have been implemented using the following implementation frameworks:

- For the Keyword Extraction process (see section 2.1), the POS tags of the words that compose the requirements were extracted by using OpenNLP, a Natural Language Processing library developed by the Apache Software Foundation (at http://opennlp.apache.org/). This library provides a POS tagger implementation, along with POS tags models trained with machine learning techniques.

- To perform the necessary SVD in Latent Semantic Indexing (see Sections 2.2 and 2.3), EJML was used. EJML is a basic linear algebra package for Java (available at https://code.google.com/archive/p/efficient-java-matrix-library/). Along with other features, this library provides an implementation of SVD.

- In the evaluation of the results of the code rankings retrieved by CACAO against the oracle, the code diffs were carried out by leveraging the DiffUtils library. The DiffUtils library is a Java open source library which provides methods that enable us to perform the necessary comparison operations between texts (at https://code.google.com/archive/p/java-diff-utils/).

For the evaluation of CACAO, we used a Lenovo E330 laptop, with a processor Intel(R) Core(TM) i5-3210M@2.5GHz with 16GB RAM and Windows 10 64-bit.

3.3 Results

Five iterations of the evaluation steps were run, with each of the five products playing the role of the new product and therefore being their code used as the oracle. Figure 6 shows two graphs that correspond to recall and precision results for CACAO when the 'Cincinnati' (solid line), 'Kaohsiung' (dashed line), 'Budapest' (discontinuous line), 'Houston' (dotted line), and 'Auckland' (crossed line) trains act as the new product.

For every requirement in the new product, CACAO generates one ranking. Each result in a ranking is composed by a method name, and the recall and precision values associated to that method. Results in each ranking are ordered by their relevance to the requirement development, determined by LSI.

Rankings can be shown with different numbers of results. For instance, a ranking showing its first result comprises the name of the most relevant method to the requirement, its recall value, and its precision value (i.e.: First result: m1 method, 2.76% recall, 63.41% precision). The same ranking, showing the first three results, comprises the names of three methods, their recall values, and their precision values (i.e.: First result: m1 method, 2.76% recall, 63.41% precision, Second result: m5 method, 3.52% recall, 72.49% precision, Third result: m8 method, 1.48% recall, 82.2% precision).

The left part of Figure 6 shows recall results of CACAO for the five new products. The horizontal axis represents the number of results shown in the rankings for all the requirements of the new product. The vertical axis represents the recall percentage, resulting from adding the recall of all the rankings generated. The formula for recall for one ranking, when k results are taken in account, is as follows:

$$Recall@k = \sum_{i=1}^{k} Recall(i) - C$$

Where C is calculated by adding the recall of the second and subsequent repetitions of the methods that have already appeared in the summation once.

For instance, a value of 37 in the horizontal axis, which returns a value of around the 26% total recall for 'Auckland' and around the 60% total recall for 'Kaohsiung' and 'Cincinnati', represents that when 37 results are shown in all the rankings, the recall of all the methods shown (not counting duplicate methods), adds up to around the 26% when the 'Auckland' train is the new product and up to around the 60% when either the 'Kaohsiung' train or the 'Cincinnati' train act as the new product.

By looking at recall results, it is possible to appreciate that the maximum recall (maximum percentage of the oracles that CACAO can cover) reaches up to the 67% for 'Cincinnati', 61% for 'Kaohsiung', 55% for 'Houston', 52% for 'Auckland', and 34% for 'Budapest', when each one is treated as the new product. In the cases of 'Kaohsiung' and 'Budapest', taking 60 results would suffice to fulfill the maximum recall, while in the case of 'Cincinnati' it would be necessary to increase the rankings size up to nearly 70 results, and more than 90 would be needed for 'Houston' and 'Auckland'. In the cases of 'Cincinnati', 'Budapest', and 'Kaohsiung', with rankings of 40 elements, around the 90% of the maximum recall would be achieved, while in the cases of 'Houston' and 'Auckland' about 80 results would be needed.

The right part of Figure 6 shows the precision results of CACAO for the five new trains. The horizontal axis represents the number of results shown in the rankings. The vertical axis represents the precision percentage associated to the results shown in the rankings, resulting from calculating the average precision of all the rankings, including duplicate methods. The formula for precision for one ranking, when k results are taken in account, is as follows:

$$Precision@k = \frac{\sum_{i=1}^{k} Precision(i)}{k}$$

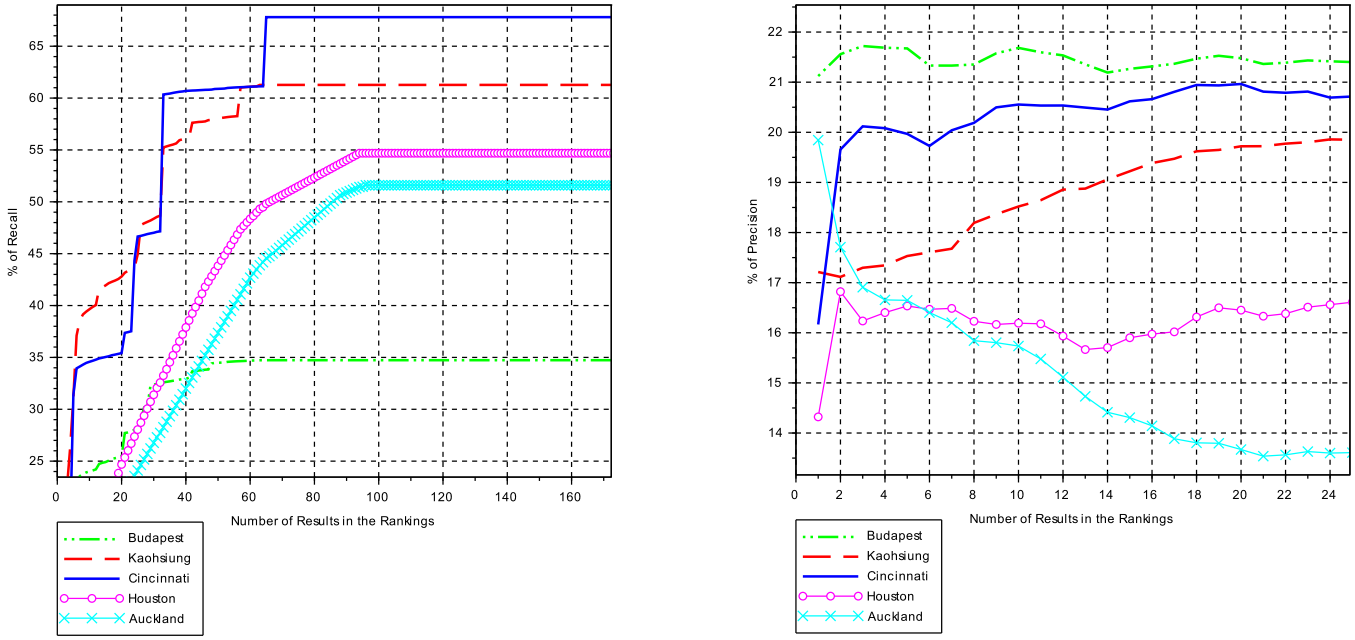

Figure 6: Recall and Precision results of the rankings

For instance, a value of 12 in the horizontal axis, which is around the 15% precision for 'Auckland' and around the 21.5% precision for 'Budapest', represents that when 12 results are shown in all the rankings, the average precision of all the rankings shown revolts around the 15% when the 'Auckland' train is the new product and around the 21.5% when the 'Budapest' train is the new product.

Putting the focus on precision results, it can be appreciated that the maximum average precision (maximum average percentage of the methods that is present in the oracles) reaches up to around the 21.7% for 'Budapest', the 21% for 'Cincinnati', the 19.9% for 'Kaohsiung' and 'Auckland', and 16.9% for 'Houston', when each one is the new product. Rankings of around 5 positions would have precision values from around the 80% to almost 90% of the total precision in all cases except 'Auckland', where precision descends as the number of positions in the rankings augments. As more positions in the rankings are taken in account, values of precision become stable.

Data shows that it is likely to find relevant code in the rankings. CACAO results show that by reviewing a reduced percentage of the products presented, enough code can be found to cover a percentage of a new product. For instance, results show that by reviewing the first 37 positions of the rankings, relevant code can be found to cover between the 26% and the 60% of the new product.

We have pondered about the number of results in the rankings that software engineers need to look at in order to achieve useful code results, and we concluded that, in practice, it will not be necessary to review 37 methods per requirement to that extent. As pointed out by the second reviewer of our work, our metrics are affected due to oracles used in our evaluation being far from optimum. With an oracle that reflected all the possible code reuse, our recall and precision would improve, thus needing less ranking positions to achieve meaningful results. Further discussion about this fact can be found in the following section. Besides, it is reasonable to think that software engineers using CACAO will stop reviewing methods after finding out the code they need.

4. DISCUSSION

By means of a Focus Group and semi-structured interviews with the software engineers, we compared their current CAO practice with the results of CACAO. The software of the five trains presented was developed by two different teams of software engineers. The two teams are geographically separated, but communicate through e-mail, periodic video-conferences, and weekly physical meetings. One of the teams (T1) developed the 'Houston' and 'Auckland' trains, while the other team (T2) developed the 'Budapest', 'Kaohsiung' and 'Cincinnati' trains.

We inquired the teams on whether they reviewed the code of the other team, and if so, on which percentage, when they develop a new product. T1 reported reviewing just a 5% of the code developed by T2, being that 5% mostly helper functions like signal delaying. T2 reported reviewing a 0% of the code developed by T1. However, the results of the CG-LSI performed by CACAO indicate that, given a train produced one team, the trains developed by the other team should be reviewed to obtain the maximum recall. In other words, given a new development by one team, the trains produced by the other team are relevant to perform CAO.

In Figure 3 it is possible to appreciate that for 'Cincinnati' (produced by T2), the most relevant train in terms of requirements is 'Houston' (produced by T1). Engineers

confirmed that, with manual CAO, the code from 'Houston' was not used in the 'Cincinnati' development, and that it would never be used for a T2 development. With CACAO, engineers in T2 are suggested to use 'Houston' and its methods for their future products, even if they were not behind its development.

Through this kind of situations, we noticed that:

1. Since there is an independence between teams, methods from products developed by one team can be false positives in the rankings for oracles developed by the other team. The products of both teams may be similar regarding the terms used, but few lines of code will be actually shared between them since no code is reused in practice. These false positives appear in the method rankings, and present low recall and precision, affecting the metrics of our approach. Finding the proper way to filter out these false positives remains as future work.

2. We are lacking an ideal evaluation scenario. The ground truth is that the oracles used through our evaluation are software products coded through manual CAO. Due to the CAO limitations mentioned throughout this work, the code of the products used as oracles is far from perfection in the reuse aspect.

   Therefore, in our evaluation, we are comparing a version of code reuse that has been designed attending to the requirements specifications with oracles that are not built in this same manner but rather on a manual fashion and relying on human factors. It is not possible for developers to perfectly discern how much code can and should be reused for the development of a new product.

   Comparing the methods extracted by our approach with scenarios that lack the ideal conditions lowers our precision and recall. Should we encounter an oracle with the ideal code reuse conditions, where all the code from legacy products that could and should be reused has been reused and modified to some extent, values of recall and precision would increase as more lines of code would be shared between the methods and the oracle.

In addition, we analyzed why 'Budapest' presents much worse recall results than the rest of the trains presented in this study when it acts as the oracle. Inspecting the code, we could note that the variables are coded with a different naming convention than the one in the rest of the products. When evaluating CACAO, on the diff performed between the methods and the oracle, code deltas that represent modifications of the code are treated as completely different lines such as new lines or deleted lines. As the train variables are named different, recall levels lower. In the light of the results, code modifications should be analyzed instead of directly discarded.

To avoid this issue, we should consider using more Natural Language Processing techniques in future developments of CACAO. For instance, stemming [19] should be used at some point of our approach. Stemming reduces words to their root. The objective is to unify words to avoid duplicity of terms. For example, 'coupling' will be stemmed to 'couple' or 'brakes' to 'brake'. This will allow us to retrieve concepts and keywords in an optimized fashion. As of today, applying this sort of techniques and analyzing their implications in our approach remains as future work.

5. THREATS TO VALIDITY

In this section we discuss some of the issues that might have affected the results of the evaluation and may limit the generalization of the results. We use the classification of threats to validity of [24, 30] to acknowledge the limitations of our approach.

Construct validity: This aspect of validity reflects the extent to which the operational measures that are studied represent what the researchers have in mind. To minimize this risk, we measured the factors of recall and precision. These measures are widely accepted in the software engineering research community [27, 26].

Internal validity: This aspect of validity is of concern when causal relations are examined. There is a risk that the factor being investigated may be affected by other neglected factors. The number of members in the family of trains may look small, but the products presented cover a wide range of railway types, from trams to medium-long distance trains. Furthermore, the products used in this study have been developed by different developer teams working for our industrial partner.

External validity: This aspect of validity is concerned with to what extent it is possible to generalize the finding, and to what extent the findings are of relevance for other cases. Software in the railway domain is representative of safety-critical systems like those present in the automotive domain or the aerospace domain. Nonetheless, CACAO should be applied to other domains before assuring its generalization.

Reliability: This aspect is concerned with to what extent the data and the analysis are dependent on the specific researcher. For our research, the data was recovered from trains chosen and provided by our industrial partner. The evaluation is performed by comparing the data with the trains themselves, acting as oracles.

6. RELATED WORK

Approaches related to the one presented in this paper comprehend feature location techniques carried out at the code level. Typechef [11] provides an infrastructure to locate the code associated to a given feature by means of analyzing the #ifdef directives. Trace analysis [7] is a run-time technique used to locate features. When the technique is executed, it produces traces indicating which parts of code have been executed.

Some approaches related to feature location use LSI to extract the code associated to a feature. Poshyvanyk et al. [20] combine a scenario-based probabilistic ranking of events and information retrieval via LSI. Given a query formulated by the user to identify the feature and two sets of scenarios (one that exercises the feature and other that do not), their system ranks the program methods using LSI. They rank each executed method based on the frequency of its appearance in the trace. Liu et al. [14] combine information from an execution trace and from the comments and identifiers from the source code. They executed a single scenario, which exercises the desired feature, and all executed methods are identified based on the collected trace using LSI.

The prior techniques have been generally applied to searching the code of a feature that has to be extended or is involved in the fixing of a bug. Our approach extends the ideas of the previous works by involving the analysis of requirements and leveraging the fact that products form a family, instead of treating them as independent items. Unlike the previous works, our approach analyzes the requirements of the family of software products to determine which are the most relevant for reuse in the scenario of a new development, and later calculate rankings of the most relevant methods in the legacy products for the implementation of each requirement in the new product.

Feature location approaches in a product family such as the one presented in [31] center their efforts in finding the code that implements a feature between the different products by combining techniques such as FCA and LSI. In our approach, we are not interested in the best representation of a feature in the family, but in locating the most relevant methods that implement a requirement (regardless of whether it represents a feature, a fragment of a feature, or several features). Since engineers must review the proposed methods to decide what to reuse, our approach also differentiates from [31] by introducing a step (Product Relevancy Analysis) where engineers decide over which products the location is made, balancing product relevancy and knowledge about the family: potentially, more code can be found on relevant products, but with a good level of knowledge of a product, it becomes easier for engineers to reuse code.

Other work [28] focuses on applying reverse engineering to the source code to obtain the variability model. In [4] the authors use propositional logic which describes the dependencies between features. In [17] the authors combine Typechef techniques and propositional logic to extract conditions among a collection of features.

These works engage explicitly the variability of the legacy products, but do not indicate the most relevant methods in the legacy products for the development of each requirement in the new product, as our work does.

7. CONCLUSIONS

To keep pace with the increasing demand for custom-tailored software systems, companies often apply the Clone-and-Own practice, through which a new product in a software product family is built by copying and adapting code from other family products. Clone-and-Own is imperfect and in industrial scenarios, it can be a time and effort-consuming process without guaranteeing good results.

In this work, we show our approach, named Computer Assisted CAO (CACAO). Given a set of natural language requirements for a new product in a software product family, and the requirements and code of the legacy products, CACAO leverages Part-of-Speech tagging and Latent Semantic Indexing to rank the most relevant products to the new development at the requirements level first, and to locate the most relevant methods to each requirement of the new product in the second place. CACAO produces, for each requirement of the new product, a ranking of the most relevant methods in the family for the development of the requirement. Software engineers can use the rankings to avoid the mentioned CAO issues.

We have evaluated our approach on the railway domain with our industrial partner, Construcciones y Auxiliar de Ferrocarriles (CAF), who provided a family of five train control software products. The results of CACAO show that it is likely to find relevant code in the rankings. Furthermore, CACAO revealed products that were not considered to be reusable by the software engineers to be relevant for code reuse, as in the case of the 'Houston' train for the 'Cincinnati' train development. Finally, as future work, we plan to apply more Natural Language Processing techniques such as stemming to avoid the issues related to different naming conventions as seen in the 'Budapest' train, which achieved the lower recall values in our evaluation.

8. ACKNOWLEDGMENTS

We would like to express our gratitude to the CAF software engineers for providing the requirements and code of the trains presented through this work, holding meetings with the authors, and reviewing earlier versions of this paper. In particular, we would like to thank Óscar Sánchez, Javier Martín, Óscar Aldana, and Elena Remacha, without whom this work would have not been possible.

We would also like to thank the anonymous reviewers of our paper for their valuable feedback and comments, which helped us improve our work.

9. REFERENCES

[1] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 67–76, Sept 2008.

[2] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, t. Stănciulescu, A. Wąsowski, and I. Schaefer. Flexible product line engineering with a virtual platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 532–535, New York, NY, USA, 2014. ACM.

[3] N. H. Bakar, Z. M. Kasirun, and N. Salleh. Feature extraction approaches from natural language requirements for reuse in software product lines. *J. Syst. Softw.*, 106(C):132–149, Aug. 2015.

[4] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, pages 23–34, 2007.

[5] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 25–34. IEEE, 2013.

[6] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli. On-demand feature recommendations derived from mining public product descriptions. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 181–190, New York, NY, USA, 2011. ACM.

[7] A. D. Eisenberg and K. D. Volder. Dynamic feature traces: Finding features in unfamiliar code. In *21st IEEE International Conference on Software*

Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary, pages 337–346, 2005.

[8] A. Ferrari, G. O. Spagnolo, and F. Dell'Orletta. Mining commonalities and variabilities from natural language documents. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 116–120, New York, NY, USA, 2013. ACM.

[9] A. Hulth. Improved automatic keyword extraction given more linguistic knowledge. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pages 216–223. Association for Computational Linguistics, 2003.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[11] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 805–824, 2011.

[12] T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.

[13] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, 2006.

[14] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 234–243, New York, NY, USA, 2007. ACM.

[15] F. Liu, D. Pennell, F. Liu, and Y. Liu. Unsupervised approaches for automatic keyword extraction using meeting transcripts. In *Proceedings of human language technologies: The 2009 annual conference of the North American chapter of the association for computational linguistics*, pages 620–628. Association for Computational Linguistics, 2009.

[16] R. T.-W. Lo, B. He, and I. Ounis. Automatically building a stopword list for an information retrieval system. In *Journal on Digital Information Management: Special Issue on the 5th Dutch-Belgian Information Retrieval Workshop (DIR)*, volume 5, pages 17–24. Citeseer, 2005.

[17] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: static analyses and empirical results. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 140–151, 2014.

[18] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering*, pages 276–286. IEEE Computer Society, 2009.

[19] M. Porter. Snowball: A language for stemming algorithms, Oct. 2001.

[20] D. Poshyvanyk, Y. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.

[21] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1233–1236, Piscataway, NJ, USA, 2013. IEEE Press.

[22] J. Rubin and M. Chechik. A survey of feature location techniques. In *Domain Engineering*, pages 29–58. Springer, 2013.

[23] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 156–160, New York, NY, USA, 2012. ACM.

[24] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

[25] H. Saif, M. Fernández, Y. He, and H. Alani. On stopwords, filtering and data sparsity for sentiment analysis of twitter. In *LREC 2014, Ninth International Conference on Language Resources and Evaluation. Proceedings.*, pages 810–817, 2014.

[26] H. E. Salman, A. Seriai, and C. Dony. Feature location in a collection of product variants: Combining information retrieval and hierarchical clustering. In *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, pages 426–430, 2014.

[27] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[28] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 461–470, 2011.

[29] C. Silva and B. Ribeiro. The importance of stop word removal on recall values in text categorization. In *Neural Networks, 2003. Proceedings of the International Joint Conference on*, volume 3, pages 1661–1666. IEEE, 2003.

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[31] Y. Xue, Z. Xing, and S. Jarzabek. Feature location in a collection of product variants. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pages 145–154, 2012.