Published in Proceedings of the 7th Forum for Information Retrieval Evaluation which should be cited to refer to this work. DOI 10.1145/2838706.2838710

Document Retrieval Metrics for Program Understanding

Eric Harth
Geneva School of Business Administration
Univ. of Applied Sciences Western Switzerland
7, route de Drize, 1227 Carouge, Switzerland
eric.harth@hesge.ch

Philippe Dugerdil
Geneva School of Business Administration
Univ. of Applied Sciences Western Switzerland
7, route de Drize, 1227 Carouge, Switzerland
philippe.dugerdil@hesge.ch

ABSTRACT
The need for domain knowledge representation for program comprehension is now widely accepted in the program comprehension community. The so-called “concept assignment problem” represents the challenge to locate domain concepts in the source code of programs. The vast majority of attempts to solve it are based on static source code search for clues to domain concepts. In contrast, our approach is based on dynamic analysis using information retrieval (IR) metrics. First we explain how we modeled the domain concepts and their role in program comprehension. Next we present how some of the popular IR metrics could be adapted to the “concept assignment problem” and the way we implemented the search engine. Then we present our own metric and the performance of these metrics to retrieve domain concepts in source code. The contribution of the paper is to show how the IR metrics could be applied to the “concept assignment problem” when the “documents” to retrieve are domain concepts structured in an ontology.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]: *Restructuring, reverse engineering, and reengineering*

General Terms
Algorithms, Measurement, Experimentation

Keywords
Document retrieval metrics, Program comprehension, Domain ontology, Dynamic analysis.

1. INTRODUCTION
Program comprehension has been a hot topic in software engineering for more than three decades with pioneering work in software psychology [12]. As early as 1983, Brooks proposed that program understanding be defined as the process of re-creating the links between the domain problem and the program code by hypothesis generation, refinement and validation [2]. As of 1995, the main theories of program comprehension for maintenance have been analyzed by Mayrhauser and Vans who proposed a program comprehension metamodel [13].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
FIRE '15, December 04-06, 2015, Gandhinagar, India
© 2015 ACM. ISBN 978-1-4503-4004-5/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2838706.2838710

The authors explained that top-down hypothesis generation should sometimes be complemented by bottom-up program analysis. These early works highlighted the need for domain knowledge to be explicitly taken into account in program understanding. This vision has since gained an increasing acceptance in the software engineering community [10]. In the mid 90's Biggerstaff, Mitbander and Webster coined the term “concept assignment" [1] to name the search and assignment of human-oriented concepts to the elements of the program code. The authors explain that during program understanding the software engineer would discover and interrelate informal human-oriented concepts step by step to build an understanding of the program (i.e. create a mental model [13,9]). In fact, this vision is close to that of Brooks [2].
In this paper we propose a new approach to the concept assignment problem by using ontologies [18] and documents retrieval metrics [19]. The concepts are then considered the “documents” to retrieve using “queries” represented by the code of the methods. After some background information given in section 2, section 3 proposes an introduction to program understanding and to our approach that represent the sequence of domain concepts involved in the execution of the program. Section 4 deal with the identification of the concepts in the source code, in particular the structure of the ontology we used and the stemming technique needed to match the strings. Section 5 deals with the metrics we used to compute the distance between the terms in the source code of the methods and the concepts. Section 6 presents the results of our concept retrieval experiments using several metrics. Section 7 concludes de paper. The annex in section 10 presents some implementation issues.

2. BACKGROUND
In the paper of Biggerstaff, Mitbander and Webster [1] the notion of “concept" is not precisely defined. Consequently, many researchers have since worked on the “concept assignment problem" while speaking about widely different things. Rajlich and Wilde recognized this problem and presented the way “concepts" can be represented in programs and the role they play in program comprehension [8]. Recently, the kinds of knowledge required by program maintenance engineers has been summarized in Maalej et al. [16] that present the current approaches in program understanding. As far as document retrieval techniques are applied to the “concept assignment problem”, Marcus proposed a method to use semantics to drive program analysis [15]. His approach is based on the retrieval of information from the source code and the associated documentation (i.e. user manual), using machine learning models and document indexing techniques. Therefore, this technique is applicable only if there is some useful and accurate documentation on the program. This is generally not the case for legacy systems. Starting from the ideas of Markus, Kuhn at al. [14] built a tool to identify clusters in source code using the latent semantic indexing (LSI) over the source artifacts. The inputs of the indexing mechanism are the identifiers and the comments in the source code. The trouble is that most of the legacy system for which program comprehension is needed lack reliable program

comments. In our research we explicitly represent domain concept as ontologies. But the key problem to solve is finding a technique to identify the references to the domain concepts in the source code of the program. The document/query metaphor borrowed from the information retrieval (IR) domain, allows us to reuse indexing methods usually applied to text or natural language representation. In this context several methods are generally used to find similarity between a list of terms and a corpus of textual documents:

- Classical string distance methods based on character comparison [5]: Manhattan, Dice, Jaccard, Levenshtein, Jaro-Winkler distances.
- Vector-based methods based on the Latent Semantic Analysis (LSA) [3] using the Single Value Decomposition (SVD) to reduce the size of the original documents without losing their internal contextual and local semantic relationship [6]. The LSA methods can be viewed as a space compression method that simplifies drastically the similarity calculation among the original query and the target documents. However, as noted by Oates et al. [7], LSA has some drawbacks and are best used in pattern recognition and data clustering.
- Statistical methods based on TFIDF frequency analysis [11,17], are easier to implement than LSA and generally quite effective. They exploit statistic terms occurrence within a document and their global usage among the corpus. The more frequently a term appears in corpus, the less discriminant it is for similarity calculation.

In our study, Jaccard, Levenshtein and TFIDF techniques have been evaluated to identify the reference to concepts in the source code.

3. CONCEPTS, CODE COMPREHENSION

Since the goal of our research is to help with the understanding of programs, we must analyze program execution. Indeed it is well known that the semantics of programs comes from the interpretation of the sequence of its "commands" [4] (i.e. the program statements). However, in our case, the interpretation must be at a higher level: in the context of the application domain. In other words, we must explain what the program does in terms of the domain concepts. While we could theoretically analyze all the paths through all the statements of the program, this is infeasible in practice for industrial size programs. To overcome the problem, one approach is to choose a set of program usage scenarios (use-cases) and analyze the code that gets executed when the scenarios are run. But we are well aware that, depending on the chosen scenario, some of the program paths may never be executed. However, our ambition is not to be able to explain every single path through the program but only the meaningful ones i.e. the ones that correspond to use cases relevant to the business. Therefore our tool analyzes the sequence of concepts that get referenced when the program runs according to some scenario. The interpretation of the (business related) meaning of the program comes from the comparison of the sequence of concept with the purpose of the scenario in the business domain. But a full explanation of our program understanding technique is beyond the scope of this article. The analysis of the running of a program is called dynamic analysis. Most often it is done off-line by analyzing a record of the sequence of methods that get executed when the scenario is run. This record is called the execution trace. There are several techniques to generate it. The one we chose is to instrument the source code of the program that consists of inserting extra statements in the source code to record events when the methods are executed. In our implementation, an event is generated when the method is entered and exited.

Besides, the source code of each method of the program is statically analyzed to identify the business concepts referenced in the methods. This allows us to graphically represent the sequence of the concepts that are referenced when the scenario is run by displaying the concept involved in each executed method. We call this representation the "Concept time series" (figure 1). On the left side we represented the two sources of information needed to generate the concept time series: the execution trace file (sequence of method signatures) and the method to concept database. The latter holds the relation between the methods and the concepts referenced in each method. Each relation is associated with a weight that represents the "strength" of the relation (i.e. how strongly the code of the method evokes the concept). Our analysis tool then merges the information from these sources to compute the concept time series presented on the right. The x axis represents time and the y axis the frequency of the use of the concepts.

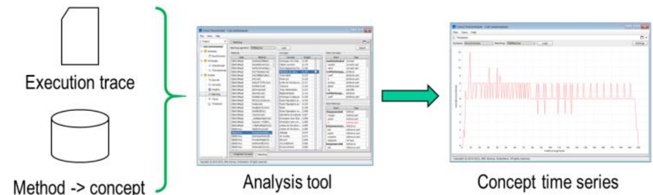

Figure 1. Concept Time Series

4. MAPPING CONCEPTS TO METHODS

To map the concepts to the methods, we must rely on clues. These are the collections of strings that are contained in the methods and in the concepts. Then, by measuring the overlap between these collections, we compute the strength of the relation from method to concept. The domain concepts are structured in an ontology i.e. an explicit representation of the concepts and the links between them. The concepts are represented by a name and a set of attributes that characterize the concept. The concepts could be linked to each other by several relations, but two of them hold a specific semantic:

- Subclass-of (ISA): the specialization link that goes from a specialized concept to a more generic one.
- Part-of: the link that goes from a concept that represents a component to the concept that represents the compound.

This concept representation is manually built with the help of an expert from the domain and is rather independent from any programs. But the developers of programs in this domain may have some special way to name the concepts and their attributes. For example there could be naming conventions in the enterprise and/or programming language constraints that may prevent the programmer from using the full names. If possible, we review the domain concept ontology with programmers to know how the concepts and their attributes may be named in the programs. This lead to an ontology having two layers:

1. The program independent layer where the concepts and attributes are named according the conventions in the domain.
2. The program specific layer which translates the strings of the first layer to strings used in the programs.

This is showed in figure 2. The strings used to describe the concepts in layer 1 are translated, in layer 2, to the strings that may be found in the programs. For example, the concept in layer 1 can be represented by program classes with some specific naming convention. Moreover, each concept attribute may correspond to

several identifiers in the programs. In particular, this is the case for legacy software which underwent several generations of maintenance programmers. Moreover, there could also be conventions to name the variables that will reference the instances of a class. These candidate names for the variables can also be recorded in layer 2. Of course, the same string in layer 2 could be mapped to the attributes of several concepts in layer 1.

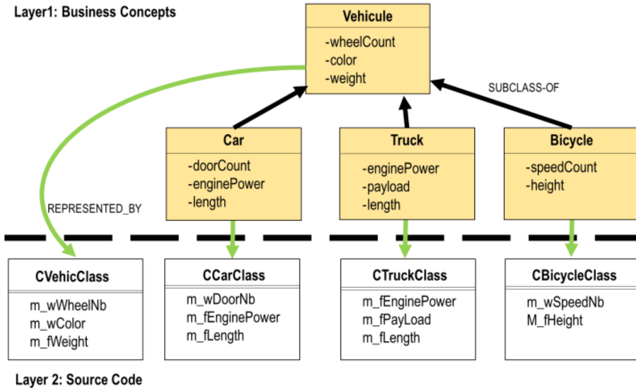

Figure 2. Business concepts ontology structure

Figure 3 displays a screenshot of our tool showing the representation of the two layers of an ontology in the domain of heat exchangers. It shows structure of a concept (“Fluide”) with its attribute names (left) and the corresponding program identifiers (right). The bottom of the screen displays, on the left, the class name representing the concept in the programs and on the right the identifiers (variable names) we may find in the programs to reference instances of this class.

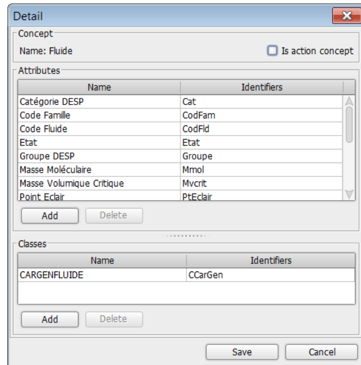

Figure 3. Layers 1 and 2 for a concept in the ontology

In summary, the collection of strings associated to a concept includes the name of the concept, the name of its attributes as well as the programming level equivalent of these strings. As for the methods, the strings are the names of: the methods, the parameters, the variables, the constants as well as the names of the types of: the parameters, the variables and the object returned by the method. Figure 4 presents the mapping between the method’s source code and the concepts that our two layers ontology enables. Beside each concept (oval) we represented a few identifiers that can be used to retrieve the references to the concepts in the source code.

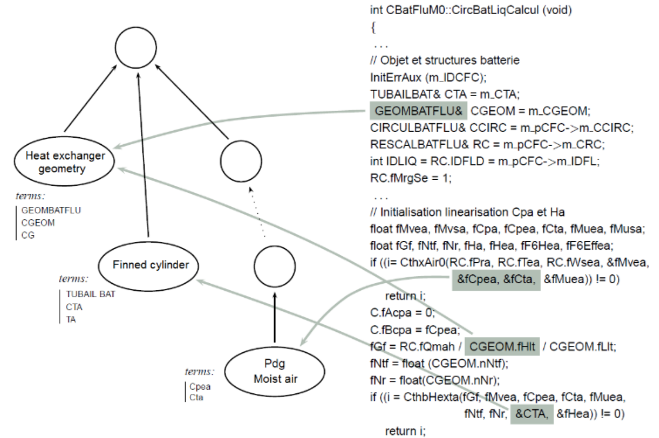

Figure 4. Mapping concepts to method’s source code

A key hypothesis in our work is that the program source code is not obfuscated and that all identifiers (methods, parameters, variables) are meaningful and carefully chosen by the programmers.

5. METRICS

5.1 Introduction

There are many techniques available to match the strings of the methods and the concepts. The simplest is to compute the size of the intersection of the collections of strings associated to the methods and the concepts. However, there are many problems with this. In particular, we know that some strings are very specific to a concept while others are very general. Therefore, the specific strings should have more weight in the matching than general ones. If we roughly apply this simple technique, several non-relevant concepts could be linked to the methods. For example, if a method contains the string “height”, all the concepts with a “height” attribute will be associated to it. Another problem is to cope with the special syntax of the strings in the programs. For example the method names are often composed of several words identified using the camel syntax. Such string would probably not correspond to any single string contained in the concepts (layer 1). Then, we must split the strings from the program into their components words before proceeding with the matching. Finally, the names could be written in the singular or plural forms on both sides. One should therefore simplify these strings to make them comparable (i.e. extract the root form or “stem” of the string). We soon realized this kind of processing to be analog to what is used in document retrieval if one considers the concepts to be the documents to retrieve and the strings in the method the element of the “query”. Then, we explored several metrics to find the one that would be best suited to the problem. Since we are dealing with document retrieval techniques, we will use the word “term” to mean any string that is relevant for the matching. The collections of terms to compare are computed using the following processes (Figure 5):

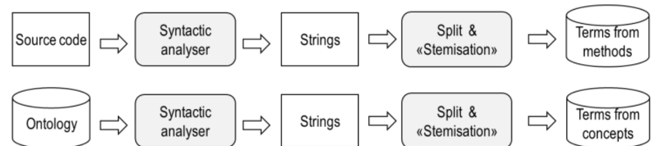

Figure 5. Production of the terms to compare

The “stemisation” a procedure used to reduce inflected or derived words to their root form [21]. As a result, a collection of root terms is associated to each method and each concept. The matcher will

then use a metric to compute the method to concept relation (figure 6).

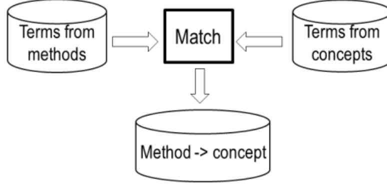

Figure 6. Term matching

5.2 TF-IDF

This is one of the most popular metrics for document retrieval [17]. If the general idea of the metric is always the same, the specific implementation may vary. Here is the way we applied it to our context. Let M be the set of methods, C be the set of concepts, tc(c) be the collection of the terms associated to the concept c, t be a term and occurrences(t,c) be the number of occurrences of t in c. Then we have:

- TF(t,c), the Term Frequency, is the number of occurrences of the term t in the concept c relative to the number of term in c: TF(t,c) = occurrences(t,c) / | tc(c) |. This technique allows the computation to be independent from the concept size (number of terms).
- IDF(t,C), the Inverse Document Frequency, computes the inverse proportion of the term t over the entire concept corpus as: IDF(t,C) = log (| C | / (1 + | { c ∈ C | t ∈ tc(c)} |))

By multiplying both measures, we get the final metric: TFIDF(t,c) = TF(t,c) * IDF(t,C), which expresses the "strength" of the term t to retrieve the concept c. Now we must gather the results for all the terms in the collections. Following a technic used in IR, for each concept c we will compute a vector in the space of the terms where each element of the vector is the value of TFIDF(t,c) for the corresponding term t. Usually, in document retrieval, to compute the "proximity" of two documents we compute the cosine of the angle between their vectors. However in our case we are not interested in the proximity of the concepts themselves but in the proximity of the methods to the concepts. Therefore we must compute a TF-IDF vector for the methods too. But for the methods the term frequency is not relevant. Indeed the number of times a term (a variable name for example) is used in a method is much less an indication of the "strength" of the term in the method (whatever it could mean) than it is a characterization of the programming style. Moreover, it is not relevant to compute the IDF for the methods because the goal is not to retrieve the methods but the concepts from the strings found in the methods. Consequently we reuse the IDF factor computed for the concepts. In summary the values of the TF-IDF metric for a method m are:

- TF(t,m) = 1 if the term is present in the method, 0 otherwise.
- IDF(t,M) = IDF(t,C)

The computation of the cosine of the angle between the vectors of a concept and a method gives the strength of the evocation of the concept by the method. If m_i is the vector of a method and c_j is the vector of a concept, the weight of the evocation (m_i, c_j) is:

$$weight_tfidf(m_i, c_j) = m_i \bullet c_j \,/\, |m_i| * |c_j|$$

5.3 Jaccard similarity

The *Jaccard similarity* is a statistical measure to compare the diversity of two sets of terms [23]. In our context, it is computed as the ratio between the sizes of the intersection and the union of the sets of terms found in the concepts and the methods. It is defined as:

$$weight_jaccard(c, m) = \frac{|tc(c) \cap tm(m)|}{|tc(c) \cup tm(m)|}$$

Where:

- c : a concept in the ontology
- m : a method in the source code
- tc(c) : collection of the terms associated to the concept c
- tm(m) : collection of the terms associated to the method m

This coefficient is usually normalized in the interval [0,1] and is interpreted as semantic proximity between a concept and a method. The greater the coefficient the more similar the concept *c* and the method *m*.

5.4 Levenshtein similarity

This is based on the calculation of the edit distance - or *Levenshtein distance* - between two string *s1* and *s2*. This distance measures the required modifications in the string *s1* (character insertion, deletion, substitution) to transform it into the string *s2*.

$$\begin{aligned} edit_dist_{s1,s2}(l_1,l_2) = \ & \textbf{If } MIN((l_1,l_2) = 0 \textbf{ Then } MAX(l_1,l_2) \\ & \textbf{Else } MIN(\ edit_dist_{s1,s2}(l_1-1,l_2)+1, \\ & \qquad edit_dist_{s1,s2}(l_1,l_2-1)+1, \\ & \qquad edit_dist_{s1,s2}(l_1-1,l_2-1)+cost(l_1,l_2)\ ) \end{aligned}$$

$$cost(l_1,l_2) = \textbf{If } s1(l_1) = s2(l_2) \textbf{ Then } 0 \ \textbf{ Else } 1$$

Where:

- s1 : first term to compare
- s2 : second term to compare
- l_1 : length of s1
- l_2 : length of s2
- s(y) : the character y of the string s
- cost(l_1,l_2) : single character match between s1 and s2.

This metric is usually normalized in the interval [0,1]:

$$leven_dist(s1, s2) = \frac{edit_dist_{s1,s2}(|s1|, |s2|)}{max(|s1|, |s2|)}$$

To convert the distance into a similarity metric, we calculate its inverse:

$$leven_sim(s1, s2,) = 1 - leven_dist(s1, s2)$$

The greater the coefficient, the more similar the terms. The *Levenshtein similarity* between a concept *c* and a method *m* is finally computed as the average of the distance computed over the Cartesian product of both collections of terms, with duplicate pairs removed:

$$weight_levenshtein(c, m) = MEAN(leven_sim(t_c, t_m)) \quad \forall t_c \in tc(c) \quad \forall t_m \in tm(m)$$

Where:

- tc(c) : collection of the terms associated to the concept c
- tm(m) : collection of the terms associated to the method m

5.5 Structural similarity

The three above mentioned metrics do not take the origin of the term into account when computing the strength of the match. By origin we mean: is it a full string or a string component, is it a class name or an attribute name, does the concept have several matching attributes or not, etc. Hence we designed a metric that takes the origin of each term into account. The metric is defined by the following rules. First, if the name of a concept is matched against a term in the method, then we are sure the concept to be referenced

(weight = 1). Second if we match the name of a class represented in the layer 2 of the ontology with a full type name in a method, then we know that the concept associated to this class name is referenced by the method (weight = 1). If we match an identifier for that class (candidate name of a variable) with a term in the method, we are also quite confident that the concept is referenced (weight = 1). However, if this identifier only matches a subterm in the method (a single word composing a longer term), the confidence is less. We heuristically use a weight of 0.8 in this case. Example, if we know that an instance of the class "PointClass" could be referenced by a variable named "pt", then the match of that string with the term "first_pt" found in the code would get a weight of 0.8. Next, if each of the attribute names of a concept (layer 1) is found in a method, we are almost sure the concept itself to be referenced by the method. However there remains some doubt because of the possible overlap in the attribute names among the concepts. An attribute is found in a method either if its name or a corresponding program identifier represented in the layer 2 of the ontology is matched to a full term in the method, or if such identifier is matched to a subterm in the method. To account for the uncertainty, the match of all the attributes of a concept would lead to a total weight of 0.8. Then if only a subset of the attributes of a concept is found in the method, we reduce the weight accordingly. Therefore each single attribute match gets a weight of 0.8* 1/#attributes in concept. Finally, the match of an attribute name could be partial if a component word only of an attribute name is matched with a subterm in a method. If all the component words of an attribute name are matched to subterms in a method, we are still unsure about a true reference to the attribute because of the overlap in the component words among the attribute names. To account for this extra uncertainty, the match of all the component words of all the attributes of a concept would lead to a total weight of 0.7. Now, if only a subset of the words of an attribute is matched, we reduce the weight accordingly. Then, each single attribute's component word match gets a weight of 0.7*1/#words in the attribute* 1/#attributes in concept. Since the identifier of an attribute in layer 2 is supposed to be the exact string to be found in a program, we do not have a rule for its partial match.

The above rules are all equally important since each one processes some specific concept reference case (through class names, variable names or attribute names). They will then cover several application contexts and programming styles. In summary, the structural similarity metric is defined by the algorithm showed in figure 7 where:

- full_name(c) return the full name of a concept
- class_names(c) return any name of a class that represents the concept in the program code
- class_identifiers(c) returns any identifier representing the name of a variable that references an instance of such class
- #attributes(c) returns the number of attributes declared in the concept c
- #components(a) returns the number of component words of the attribute a
- full_name(a) return the full name of an attribute
- attr_identifiers(a) returns any identifier representing the attribute a in the program code.

6. RESULTS

6.1 Case Study

To evaluate the performance of the metrics in the retrieval of the concepts references in the methods we ran a set of experiments on the three projects listed in table 1. The first two projects are open source projects and the last one is an old program written by our industrial partner.

```
Algorithm:
weight_struct(c,m) = 0.
For a given concept c and method m
  If (full_name(c) OR class_names(c) OR class_identifiers(c)) is
     matched to a full term in m then weight_struct(c,m) = 1
  Else
    If class_identifiers(c) is matched to a subterm in m
    Then weight_struct(c,m) = 0.8
    Else
      For each attribute a of c
        If (full_name(a) OR attr_identifiers(a)) is matched to a full
           term in m OR if attr_identifiers(a) matches a subterm in m
        Then weight_struct(c,m) =
                 weight_struct(c,m) + 0.8 * 1/#attributes(c).
        Else for each component word w of a
          If the word is found in m
          Then weight_struct(c,m) = weight_struct(c,m)
                  + 0.7*1/#components(a)* 1/#attributes(c).
          EndIf
        EndIf
      EndFor
    EndIf
  EndIf
EndFor
```

Figure 7. Structural similarity metric algorithm

This is the project we help it understand. It was originally written in Fortran and was later translated to C / C++. Therefore the structure of the program is awkward and not really object oriented. It is therefore very difficult to understand for non-specialists (an even difficult for specialists of the domain).

Table 1 – Project in the case study

Project	Description	Lang.	LOC
Genetics	Program trying to retrieve an arbitrary DNA sequence using a genetic algorithm	Java	403
JHotel	Program managing hotel reservations and customers	Java	21475
Devor	Program computing heat exchanger geometry and constraints	C/C++	100949

6.2 Estimation of Metrics Relevance

As it is common in IR field, we will use the *precision* and *recall* factors to measure relevance of the concepts found in the methods. The *precision* factor expresses the ratio of the retrieved concept that are corrects i.e. the ones that an expert would retrieve by hand. A precision factor of 1 would mean that all retrieved concepts are correct concepts (but there could be more correct concepts than retrieved). The *recall* factor is the ratio of the correct concepts that are retrieved. A recall factor of 1 would mean that all correct concepts are retrieved (but we may have retrieved more concepts some of which are not correct). To identify the correct concepts, we did the work manually through manual code inspection: we identified all the concepts truly referenced in the method's source code.

$$precision(m,w) = \frac{|V(m) \cap T(m,w,\alpha)|}{|T(m,w,\alpha)|}$$

$$recall(m,w) = \frac{|V(m) \cap T(m,w,\alpha)|}{|V(m)|}$$

Where:

- m = the method considered
- w = the metric considered
- α = the metric threshold to keep a retrieved concept
- $V(m)$ = set of correct concepts referenced in *m.*
- $T(m, w, \alpha)$ = set of concepts retrieved in m with metric w and threshold α

We can now apply both measures to a project (where n is the number of methods):

$$precision(w) = \frac{\sum_{i=1}^{n} precision(m_i, w)}{n}$$

$$recall(w) = \frac{\sum_{i=1}^{n} recall(m_i, w)}{n}$$

And finally combine them with the unique F-measure [24]:

$$F_\beta(w) = \frac{(1+\beta) * precision(w) * recall(w)}{(\beta^2) * precision(w) + recall(w)}$$

Where:

- β = weight of recall to precision

To compare the metrics we computed the F-measure with β = 1 (*F1* measure) for each of the projects and metrics with α = 0.2 (Table 2), 0.4 (Table 3), 0.6 (Table 4). The performance of each metric as a function of the project is presented in figures 8 to 10.

Table 2 - Threshold 0.2

Project	Metric	Precision	Recall	F1-measure
Genetics	TFIDF	0.80159	0.74603	0.77281
	Jaccard	0.71429	0.75397	0.73359
	Levenshtein	0.62698	0.59524	0.61070
	Structure	0.71429	0.71429	0.71429
JHotel	TFIDF	0.59322	0.58898	0.59109
	Jaccard	0.59322	0.54379	0.56743
	Levenshtein	0.42542	0.43362	0.42948
	Structure	0.68927	0.71328	0.70107
Devor	TFIDF	0.50467	0.50467	0.50467
	Jaccard	0.47819	0.46885	0.47347
	Levenshtein	0.89252	0.89252	0.89252
	Structure	0.45327	0.45327	0.45327

Table 3 - Threshold 0.4

Project	Metric	Precision	Recall	F1-measure
Genetics	TFIDF	0.76190	0.44444	0.56140
	Jaccard	0.54762	0.37302	0.44376
	Levenshtein	0.14286	0.08730	0.10837
	Structure	0.69048	0.47619	0.56365
JHotel	TFIDF	0.69774	0.67090	0.68406
	Jaccard	0.52542	0.51695	0.52115
	Levenshtein	0.49153	0.49153	0.49153
	Structure	0.69576	0.71328	0.70441
Devor	TFIDF	0.48131	0.48131	0.48131
	Jaccard	0.46262	0.46262	0.46262
	Levenshtein	0.46262	0.46262	0.46262
	Structure	0.37912	0.49361	0.42885

Table 4- Threshold 0.6

Project	Metric	Precision	Recall	F1-measure
Genetics	TFIDF	0.66667	0.35714	0.46512
	Jaccard	0.23810	0.17460	0.20147
	Levenshtein	0.09524	0.03968	0.05602
	Structure	0.69048	0.47619	0.56365
JHotel	TFIDF	0.52542	0.51412	0.51971
	Jaccard	0.49153	0.49153	0.49153
	Levenshtein	0.49153	0.49153	0.49153
	Structure	0.69052	0.70833	0.69931
Devor	TFIDF	0.46729	0.46729	0.46729
	Jaccard	0.46262	0.46262	0.46262
	Levenshtein	0.46262	0.46262	0.46262
	Structure	0.37912	0.49361	0.42885

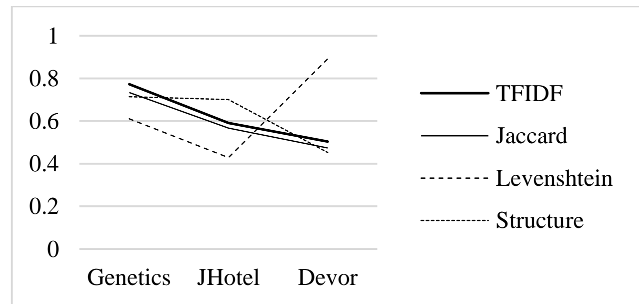

Figure 8. F1 measure, threshold 0.2

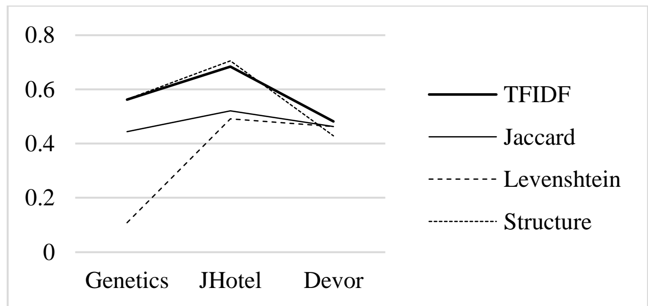

Figure 9. F1 measure, threshold 0.4

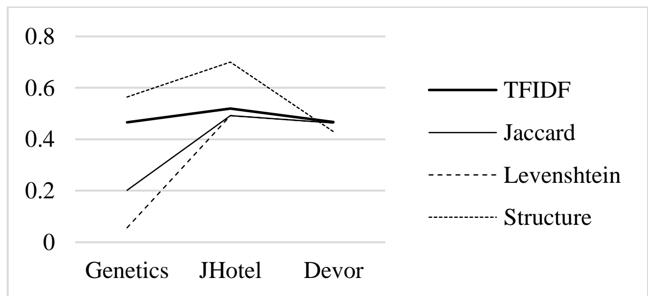

Figure 10. F1 measure, threshold 0.6

If we put aside the special result for Levenshtein in the Devor project with threshold = 0.2, that we must further investigate, the interpretation of the results are the following. Figure 8 to 10 show that our specific "Structure" metric at least equals but often outperforms all the other metrics on all projects whatever the threshold. This is due to the special fit of the metric to the semantics

of the “documents” to retrieve. We can also note that TFIDF works better than Levenshtein or Jaccard. An interesting finding is that all metrics seem to converge to about the same value whatever the threshold for the Devor project. But Devor is by far the largest project by the number of lines of code, 5 times bigger than JHotel and 250 times bigger than Genetics. So far, we do not understand if this convergence is an effect of the size or if this is driven by some other specific characteristics of Devor. But project size is clearly a candidate hypothesis since JHotel, which is 50 times bigger than Genetics, seems to show a much smaller spread of values for the metrics (for threshold > 0.2) than Genetics. The poor performance of Levenshtein does not come as a surprise since the metrics apply to the sets of “stemmized” words (all variations of a particular word have been removed). Then the comparison of two words is either 0 or large. Then the precision should be lower in comparison to the other metrics, which is mostly the case.

7. CONCLUSION

In this paper we explore the use of information retrieval approaches and metrics to the “concept assignment problem” [1]. We saw that the best performance is reached when we use our specific metric, the “Structure” metric, that takes the syntactical category of the terms (concept name or identifier, attribute name or identifier) into account. However we consider that “Structure” misses the discriminatory power of the IDF factor. Indeed with “Structure” all the terms within their syntactic category are considered equally relevant to identify some concept. But a term that is present in only one concept is much more relevant to identify this concept than those which are present in several concepts. Then, we think that a combination of the Structure metric with the IDF factor may further improve the performance of the concept assignment to the methods. This is what we will investigate in the future. Finally, the Concept Time Series that the concept retrieval metrics allows to display (§10.3) is a powerful tool to investigate the patterns of concept invocations when a program runs. By identifying these patterns we can further “summarize” the information of the execution trace hence to generate some abstract explanation of the implementation of the scenario.

8. ACKNOWLEDGEMENT

This work has been developed in the context of the Ontoreverse project supported by the Interreg IV program.

9. REFERENCES

[1] T. J. Biggerstaff and B. Mitbander. 1994. Program understanding and the concept assignment problem. *Communications of the ACM, 37(5)*.

[2] R. Brooks. 1983. Towards a theory of the comprehension of computer programs. *Intl. J. of Human-Computer Studies, 18(6)*.

[3] Dumais S.T., Furnas G.W., Landauer T.K., Deerwester S., Harshman R. 1988. Using Latent Semantic Analysis to Improve Access to Textual Information. *Proc of the ACM CHI*.

[4] Floyd R.W. 1967. *Assigning Meaning to Programs*. In: Schwartz J.T. Editor, Mathematical Aspects of Computer Science - American Mathematical Society.

[5] Moreau E., Yvon F., Cappé O. 2008. Robust Similarity Measures for Named Entities Matching. *Proc. of the Int. Conf. on Computational Linguistics.*

[6] Maletic J.I., Marcus A. 2001. Supporting Program Comprehension Using Semantic and Structural Information. *Proc. of the IEEE ICSE.*

[7] Oates T., Bhat V., Shanbhag V. 2002. Using Latent Semantic Analysis to Find Different Names for the Same Entity in Free Text. *Proc. of the ACM WIDM*

[8] V. Rajlich and N. Wilde. 2002. The role of concepts in program comprehension. *Proc. of the IEEE Workshop on Program Comprehension*.

[9] S. Rugaber. 1995. *Program comprehension*. Encyclopedia of Computer Science and Technology, 35(20).

[10] S. Rugaber. 2000. *The use of domain knowledge in program understanding*. Annals of Software Engineering, 9(1-4).

[11] Salton G., Buckley C. 1988. On the Use of Spreading Activation Methods in Automatic Information Retrieval. *Proc. of the ACM SIGIR.*

[12] Shneiderman B. 1980. *Software psychology: human factors in computer and information systems*. Winthrop Pub.

[13] von Mayrhauser A. , Vans. A. M. 1995. Program comprehension during software maintenance and evolution. *IEEE Computer, 28(8)*.

[14] Kuhn A., Ducasse S., Girba T. 2005. Enriching Reverse Engineering with Semantic Clustering. *Proc. of the IEEE WCRE*.

[15] Marcus A. 2004. Semantic Driven Program Analysis. *Proc of the IEEE ICSM*.

[16] Maalej W., Tiarks R., Roehm T., Koschke R. 2014. On the Comprehension of Program Comprehension. *ACM Trans. on Software Eng. and Methodology, 23(4)*.

[17] Ramos J. 2003. Using TF-IDF to Determine Word Relevance in Document Queries. *Proc. of the 1st Instructional Conference on Machine Learning.*

[18] Horrocks I. 2008. Ontologies and the Semantic Web. *Communications of the ACM 51(12).*

[19] Nguyen V.T., Sallaberry C., Gaio M. 2013. Mesure de la similarité entre termes et labels de concepts ontologiques. (« Measurement of similarity between terms and labels ontological concepts”. Paper in french). *Proc of the Conf. en Recherche d'Information et Applications.*

[20] Schorn M. 2009. Using CDT APIs to programmatically introspect C/C++ code. *EclipseCon, Santa Clara, California*.

[21] Porter M.F. 1997. *An algorithm for suffix stripping.* Readings in information retrieval. Morgan Kaufmann Pub.

[22] OWL API. 2015. *A Java API and reference implementation for creating, manipulating and serializing OWL Ontologies*. http://owlapi.sourceforge.net/. Retrieved August 2015.

[23] Hadjieleftheriou M., Srivastava D. – 2010. Weighted Set-Based String Similarity. *IEEE Data Eng. Bull.33(1).*

[24] Magdy W., Jones G.J.F. 2010. A Score Metric for Evaluating Recall-Oriented Information Retrieval Applications. *Proc. of the ACM SIGIR*.

10. ANNEX: IMPLEMENTATION ISSUES

10.1 Building Models

We have built a dedicated tool to record the ontology of domain concept because current open source systems do not let us attach

program identifiers to concepts (i.e. they miss layer 2 representation). To be compatible with standard tools on the market, our models can be exported in several standard formats (OWL/RDF, Turtle, Manchester and Functional Syntax). To implement this feature we are using the *OWLAPI* [22] library.

Finally, to extract the information from the source code of the programs we use the CDT/JDT frameworks taken from the Eclipse environment [20]. These let us parse the code and build an abstract syntax tree (AST) representing all source information to analyze further.

10.2 Making Strings Comparable

As the method to concept relationship problem has been reduced to term comparison, we have implemented a concept term extractor and a method term extractor to generate all the terms found in both the concepts and the methods (see figure 5). Basically, the extractors work according to the following steps:

1. Each full string is cleaned by removing all the non-alphanumeric character;
2. The type prefix or any other prefix associated to programming language conventions such as "m_" in C++ is removed;
3. The full string is split with respect to the camel-casing notation;
4. The full string and its parts are "stemmized" using the snowball algorithm [21].

This is true for all the strings but the ones declared in the layer 2 of the ontology (figure 2). For the latter the first two steps only are applied since they represent strings that are supposed to be found directly in the programs. As an example, figure 11 presents the analysis of the concept *Car* taken from figure 2. The extractor produces the terms listed in the left column of the right pane with the syntactical category of the terms (origin) listed in the right column.

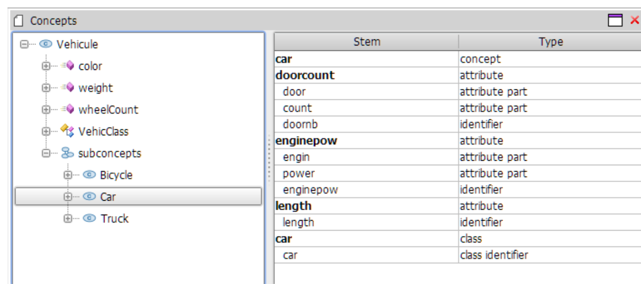

Figure 11. Result of the term extraction

10.3 Comparing term sets

To allow experimenting with different similarity metrics, and study the relevance of the concept/method matching, we implemented a view which presents the detailed results of the matching (figure 12). In this view we can select the metric to use to compute the strength of the evocation. When a method is selected in the list displayed on the left, the matching concepts are displayed in the list in the center together with their weight (strength of the evocation) and a check box ("Validated") allowing an expert to validate the match. By checking it, the expert would confirm that the concept is truly referenced in the method. This is yet another way to check the relevance of the metric. In the right part of the screen we list the set of terms associated to the selected method (top) and selected concept (bottom).

The term that match are highlighted (red). This allows us to see all the terms that are involved in the evocation of the selected concept by the selected method and allows the traceability **of** the calculations of the weight.

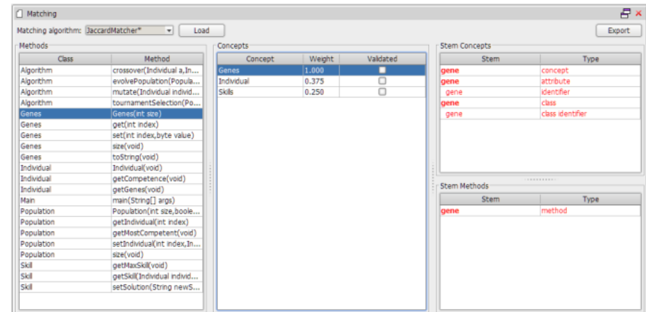

Figure 12. Detailed view of the matching terms

10.4 Building the time series of concepts

Thanks to the retrieval of the concepts in the methods, we can now display these concepts sequentially according to the method sequence of a specific scenario (execution trace). Technically, because of the huge number of methods in an execution trace, we segment the trace as adjacent segments and compute the number of times a given concept is referenced within each segment. The time series is the graphical representation of the number of evocations of the concepts in each segment as a function of time. When a user wants to generate the concept time series for a set on concept, he selects the concept in a list and set the minimal "strength" of the concept evocation by the methods. Figure 13 presents the concept time series for the Genetics project with metric = TFIDF and threshold = 0.2. In particular, we can observe the emergence of a pattern of invocations, repeated three times in the executed scenario. This can be exploited for program understanding.

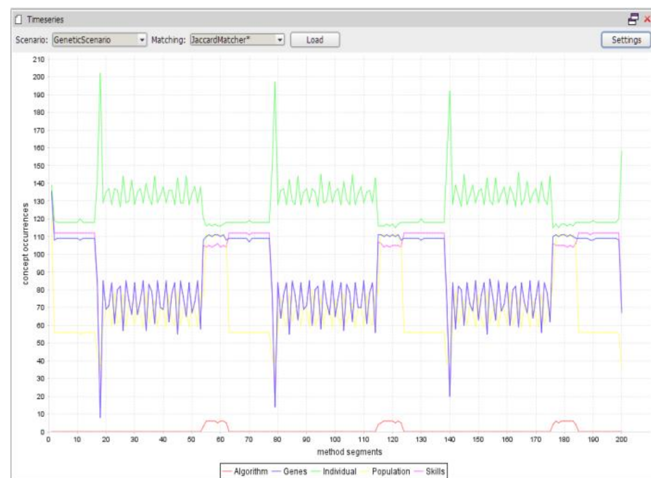

Figure 13. Concept time series with emergent patterns