Conclave: Ontology-Driven Measurement of Semantic Relatedness between Source Code Elements and Problem Domain Concepts

Nuno Ramos Carvalho[1], José João Almeida[1], Pedro Rangel Henriques[1], and Maria João Varanda Pereira[2]

[1] University of Minho, Braga, Portugal
{narcarvalho,jj,prh}@di.uminho.pt
[2] Polytechnic Institute of Bragança, Bragança, Portugal
mjoao@ipb.pt

Abstract. Software maintainers are often challenged with source code changes to improve software systems, or eliminate defects, in unfamiliar programs. To undertake these tasks a sufficient understanding of the system (or at least a small part of it) is required. One of the most time consuming tasks of this process is locating which parts of the code are responsible for some key functionality or feature. Feature (or concept) location techniques address this problem.

This paper introduces Conclave, an environment for software analysis, and in particular the Conclave-Mapper tool that provides a feature location facility. This tool explores natural language terms used in programs (e.g. function and variable names), and using textual analysis and a collection of Natural Language Processing techniques, computes synonymous sets of terms. These sets are used to score relatedness between program elements, and search queries or problem domain concepts, producing sorted ranks of program elements that address the search criteria, or concepts. An empirical study is also discussed to evaluate the underlying feature location technique.

1 Introduction

Reality shifts, bug fixes, updates or introducing new features often require source code changes. These software changes are usually undertaken by software maintainers that may not be the original writers of the code, or may not be familiar with the code anymore. In order to carry out these changes, programmers need to first understand the source code [43]. This task is probably the main challenge during software maintenance activities [9].

Software reverse engineering is a process that tries to infer how a program works by analyzing and inspecting its building blocks and how they interact to achieve their intended purpose [8,30]. Many of these techniques rely on mappings between human oriented concepts and program elements [35]. These are often used to locate which parts of the program are responsible for addressing specific

B. Murgante et al. (Eds.): ICCSA 2014, Part VI, LNCS 8584, pp. 116–131, 2014.

domain concepts [3], and are usually referred in the literature as feature location techniques [12].

Programming languages unambiguous grammars limit the sentences that can be used to write software. Still some degree of freedom is given to the programmer to use natural language terms (e.g. program identifiers, constant strings or comments). These terms can give clues about which concepts the source code is addressing, and the meaningfulness of these terms can have a direct impact on future program comprehension tasks [24]. Most of the programming communities promote the use of best practices and coding standards that usually include rules and naming conventions that improve the quality of terms used (e.g. the *"Style Guide for Python Code"* [1]). Feature location techniques that exploit such elements are typically described as textual analysis, often combined with static analysis [12].

This paper introduces Conclave[2], a system of tools for software analysis, with special focus on Conclave-Mapper[3], a tool that provides a technique to measure semantic relatedness between source code elements, and elements supplied by the maintainer as query searches. It allows the creation of mappings between source code and real world concepts, facilitating feature location activities. The main goal of this system is to provide programmers with insight and information about software packages to enhance program understanding activities and ease software maintenance tasks.

Conclave-Mapper uses source code static analysis to extract data from source code (e.g program identifiers, function definitions). The extracted data is loaded to an ontology that represents the program. Other ontologies can be added to the system if available (e.g. the problem domain ontology, dynamic traces information). Using a set of Natural Language Processing (NLP) techniques and textual analysis, *kind-of* Probabilistic Synonymous Sets (kPSS) are computed for every element present in the ontologies, and a scoring function is used to measure the semantic relatedness[4] between them. The main output of this tool are ranks – sorted by relevance – of program elements that are prone to address some specific real world domain concept. This tool also provides a Domain Specific Language (DSL), for writing search queries.

In the next section related work, and some state-of-the-art feature location techniques are discussed; Section 3 gives a brief overview of the Conclave environment; and Section 4 describes in more detail the Conclave-Mapper tool. Section 5 describes the experimental validation held to evaluate the feature location technique, including results discussion. Finally, Section 6 includes some final remarks and trends for future work.

[1] `http://www.python.org/dev/peps/pep-0008/` (Last accessed: 29-01-2014).
[2] `http://conclave.di.uminho.pt` (Last accessed: 27-01-2014).
[3] `http://conclave.di.uminho.pt/mapper` (Last accessed: 27-01-2014).
[4] In ontologies the term similarity is used to refer how similar two concepts are, and is usually based on a hierarchy of *is-a* relations, in the context of this work concepts can be related in many ways, hence the adoption of the term relatedness.

2 Related Work

Program Comprehension (PC) is a field of research concerned with devising ways to help programmers understand software systems. In the context of PC, feature (or concept) location is the process of locating program elements that are relevant to a specific feature implementation. This is typically the first step a programmer needs to perform in order to devise a code change [3, 28, 35].

Feature location techniques are usually organized by types of analysis: (a) dynamic analysis, which is based in software execution traces, and examines programs runtime (e.g. [1, 13, 40, 44]); (b) static analysis, based on static source code information, such as slicing, control or data flow graphs (e.g. [7,27,39]); and (c) textual analysis, explore natural language text found in programs like comments or documentation. This last type can be based on Information Retrieval (IR) methods (e.g. [4,5,26]), NLP (e.g. [18,41]), or pattern matching (sometimes also referred as grep-*like*) based approaches (e.g. [14]). For more details about different trends and other approaches please refer to surveys [12] and [44].

The Conclave-Mapper underlying feature location technique uses a combination of static and textual analysis, and ontologies. Examples of other approaches that explore the same combination of analysis include: in [47], Zhao *et al* use a static representation of the source code named BRCG (branch-reserving call graph) to improve connections between features and computational units gathered using an IR technology; in [17], Hill *et al* present a technique that exploits the program structure and also program lexical information; in [36], Ratiu and Florian establish a formal framework that allows the classification of redundancies and improper naming of program elements, which is used as a based to represent mappings between the code and the real world concepts in ontologies; in [16], Hayashi *et al* proposed linking user specified sentences to source code, using a combination of textual and static analysis domain ontologies. Other applications of ontologies in software engineering in [15].

State-of-the-art feature location approaches involve combining techniques taking advantage of having data produced from different types of analysis (e.g. [23, 26]).

3 Conclave Overview

The Conclave environment provides a set of tools to perform software analysis. The main system workflow is divided in three main stages: (a) collecting data; (b) processing collected data and loading ontologies; and, (c) reasoning about data in the ontologies and providing views of computed information. Figure 1 illustrates this workflow, and the next sub-sections describe in more detail the different stages. All the tools implemented in the context of this system are modular (or work as plugins), and some provide web-services, so that they can be used as standalone applications, or composed together to create more complex applications or workflows (like the one illustrated in Fig. 1).

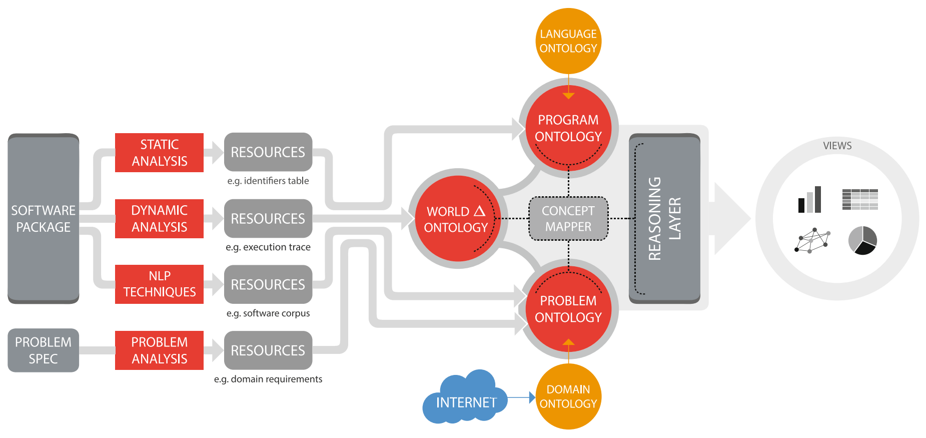


Fig. 1. Overview of the major stages of the Conclave system workflow

3.1 Collecting Data

This is the first stage of the main workflow; its goal is to collect data from a software package, and any kind of problem specification if available. It takes as input the complete package (and other available documents) and produces as output an heterogeneous collection of resources. The processing tools involved in this stage can use different type of analysis: static source code analysis (e.g. parsing code to extract identifiers and static call graphs), dynamic analysis (e.g. execution traces), etc.

Any analysis can be used to collect information, and produce a resource. In the context of this work, some tools were implemented to provide some initial data to the system and contribute to PC in general, here are some examples:

Conc-clang: is a static analysis tool, based on the clang compiler library [22] for gathering identifiers and static functions calls information for `C/C++` programs;

Conc-antlr: is a static analysis tool, based on the ANTLR parser generator framework [31], for gathering program identifiers information for `Java` programs;

Conclave-IdsProcessor: provides a tool for splitting program identifiers, mainly because programmers tend to use abbreviations and word combinations to name program elements (like variables or functions), these need to be split and expanded to improve feature location techniques [10, 11].

The heterogenous set of tools used during this stage produce a multitude of resources in distinct formats. In order to take advantage of all these results all this information needs to be conveyed to a common format, more suitable for querying and processing. Ontologies were adopted as a common target format. Building ontologies from collected data is done during the second stage, which is discussed in the next section.

3.2 Normalizing Information, Populating Ontologies

The main goal of this stage is to convey the data collected during the previous stage into the system ontologies. The input of this stage is a collection of resources, and the output is a set of populated ontologies. Usually three ontologies are populated for each software package:

Program Ontology: abstract representation of some key program elements (e.g. methods, functions, variables, classes);
Problem Ontology: concepts and relations in the application domain;
World $\triangle$ Ontology: runtime effects of executing the program (e.g. program run traces).

There are two important details about this stage. The first one is the format and technology chosen to store the ontologies. A RDF based triple-store technology was adopted to store the data. This allowed for a scalable and efficient method for performing storing and querying operations, and also allows to export the data in several community accepted ontology formats (e.g. OWL, RDF/XML, Turtle) [19, 21]. Querying facilities are also readily available; for instance, SPARQL is a querying domain specific language for RDF triple-stores [32, 34].

Although these technologies provide scalable and efficient environments for handling information, development wise, they are far from the abstraction desired by the applications level implementation. To overcome this problem the Ontology ToolKit (OTK)[5] was implemented, which provides an abstraction layer on top of the RDF technology, to develop ontology-*aware* applications. In practice, when applications developers want to perform an ontology related operation, instead of using triple-store low level primitives, they can use the abstraction layer. To motivate for the development of this abstract framework, consider the modern Object-Relational Mappers (ORM) in the context of relational databases. Which provide an abstraction layer and interface for programming languages to handle data (stored in databases) as objects, allowing the development of applications regardless of the underlying database technology used [20].

The second important detail is the data semantic shift. Resources tend to produce raw data, but the data stored in the ontologies conveys a richer semantic. Most resources require a specific tool to read the resource data, and translate it to information that is ready to store in the ontology, i.e. follows the semantic defined by the ontology. OTK has also proven useful to implement this family of tools.

A simple example to illustrate the previously discussed details follows. Imagine the CONC-CLANG tool was used to process a `C` source code file, included in a software package. The raw output of this tool is something like[6]:

```
Function,source.c::add::6,add,,source.c,6,8
```

[5] Implemented as a set of libraries for the Perl programming language.
[6] For more examples please visit the tool website: `http://conclave.di.uminho.pt/clang` (Last accessed: 27-01-2014).

This line by itself conveys small to none semantic of the data being included in the final resource. In loose english this line states that: *"in the 'source.c' file there is a 'Function' definition which has a identifier represent by 'add' that starts in line '6' and ends in line '8'"*, and this is the kind of semantic that needs to be conveyed to the ontological program representation of the program. The Program Ontology has a class to represent instances of elements that are functions in the source code, another for identifiers, and the line numbers are stored as data proprieties [7]. To illustrate the use of OTK, the following snippet illustrates a simplified version of the required code to load this information to the Program Ontology.

```
use OTK;
my $ontology = OTK->new($pkgid, 'program');
$ontology->add_instance('add', 'Function');
$ontology->add_instance('add', 'Identifier');
$ontology->add_data_prop('add', 'hasLineBegin', 6, 'int');
$ontology->add_data_prop('add', 'hasLineEnd', 8, 'int');
$ontology->add_obj_prop('add', 'inFile', 'source.c');
```

The Program Ontology used is in line with other authors' proposed descriptions (e.g. [37, 45, 46]). This also eases future integration processes with other tools that followed those same approaches. Figure 2 illustrates a subset of the class hierarchy exported to OWL. Once all the data is stored in the ontologies, the reasoning layer can be used to relate informations gathered from different

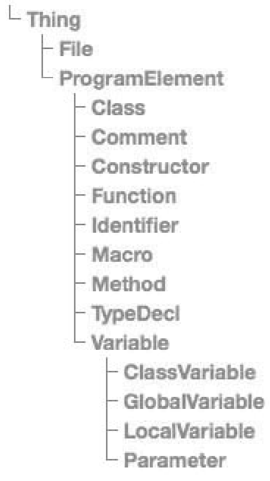


Fig. 2. Program ontology sub-set of the class hierarchy

[7] Although a triple-store RDF approach is used to store the actual information, we are using OWL vocabulary and specification to make clear the aimed semantics for the program representation [2].

elements and domains to build semantic bridges between elements. More details about this stage are discussed in the next section.

3.3 Reasoning and Views

During this stage more knowledge about the system is build and provided to the system end-user. The tools in this stage use as input the ontologies built during the previous stage, and generally fall in one of the two categories, either they: (a) process information to compute new information and knowledge about the system – usually in this case the tool output is new content added to the ontologies; or (b) information or knowledge suitable for visualization is built – in this particular case the final output of the tool is a view for the package system.

Querying the ontology, and adding information if necessary, can easily be done using the OTK framework. Also note that the tools in this stage are language agnostic, in the sense that data about the source code (language dependent) has already been gathered, and OTK tools do not depend anymore on the source language. For example, if a tool processes identifiers, to get a list of the program identifiers simply query the Program Ontology using OTK, as follows:

```
use OTK;
my $ontology = OTK->new($pkgid, 'program');
my @identifiers = $ontology->get_instances('Identifier');
```

Conclave-Mapper, described in detail in the next section, is an example of tools that are used during this stage.

4 Conclave-Mapper Feature Location Approach

Conclave-Mapper is an application that relies on data computed by other tools (see Sec. 3.1 and 3.2), to create relations between elements of any of the ontologies available for a given package. The input for this application is a set of ontologies, and either a search query, or a mapping query; and the output is a sorted rank of element relations, or a mapping of element relations respectively. The Program Ontology represents the elements of the program, a software maintainer can ask the application to compute the relations between elements in the program and either a set of keywords provided in a search query, or elements in other ontologies (e.g. Problem Ontology) using a mapping query. In the first case, the result is a sorted rank of the program elements that are related with the keywords provided in the search query, and in the latter a matrix of relatedness score between the elements selected from both ontologies. Both can be used to find which parts of the code are responsible for implementing a domain concept – feature location.

Before discussing function implementations, a formal description of the data types required for creating each output follows[8]:

[8] *Haskell* syntax is used to describe the data types and functions discussed; some details have been slightly simplified to improve readability.

```
type Rank  = [Entry]
data Entry = Entry { score :: Float, element :: Element }
```

This defines a *Rank* as a collection of *Entry*, where each *Entry* contains the semantic relatedness (*score*), between the element (*element*) and the query search (more details about how this score is calculated in Sec. 4.2). The type *Element*, represents an instance in any ontology (if elements of the Program Ontology are being used all other data is also available: source file, begin and end line, identifier, etc.).

```
data Map = Map { rows :: [Element], cols :: [Element], cells :: [Cell] }
data Cell = Cell { row :: Int, col :: Int, score :: Float }
```

This defines a *Map* as a matrix, with an *Element* for each row and column; each *Cell* in the matrix (besides its position information) contains the semantic relatedness measure (*score*) for the corresponding elements (more details on how score is calculated in Sec. 4.2).

The application implements two main functions to compute each one of the available output types. The *locate* function creates a *Rank* and is defined as:

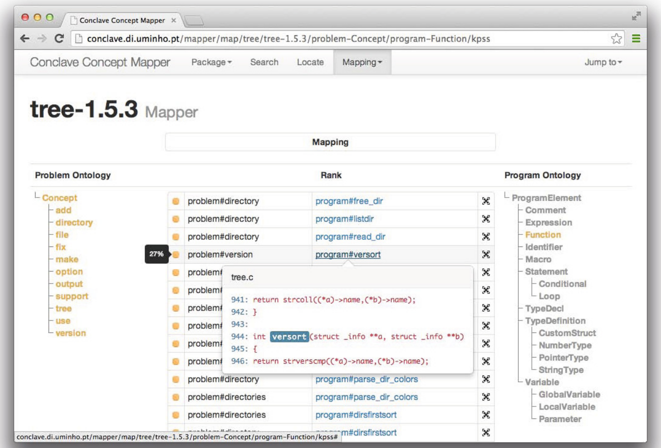
```
locate   :: Query → Rank
locate q = let elements = getElements q
               entries  = [ Entry score e | e ← elements, computeScore q e ]
           in Rank entries
```

This function, given a *Query*, computes a *Rank*, by iterating over all the elements being analyzed (defined by the search query), and for each element computing a semantic relatedness score, and adding it to the *Rank* as a new *Entry*. The element set being searched and the scoring function are defined by the search query (see Sec. 4.2 for details).

The *mapping* function creates a *Map* and is defined as:

```
mapping          :: Query → Query → SFunction → Map
mapping q1 q2 f = let rows  = getElements q1
                      cols  = getElements q2
                      i     = 0
                      j     = 0
                      cells = [ Cell i j (f r c) |
                                    r ← rows, c ← cols,
                                    i ← i + 1, j ← j + 1 ]
                  in Map rows cols cells
```

This function, given two queries (of type *Query*), and a scoring function (*SFunction*), calculates a matrix of elements where each cell includes the relatedness score between the corresponding row and column element. This provides a matrix of relations between all selected (program, application domain, etc.) elements, that can be sorted by relevance. Figure 3 illustrates a possible view of these mappings, highlighting the best relevance ranking between the application domain and functions.



problem#directory	program#free_dir
problem#directory	program#listdir
problem#directory	program#read_dir
problem#version	program#versort
problem#	
problem#	
problem#	
problem#	
problem#	
problem#	
problem#directory	program#parse_dir_colors
problem#directories	program#parse_dir_colors
problem#directories	program#dirsfirstsort
	m#dirsfirstsort

tree.c

```
941: return strcoll((*a)->name,(*b)->name);
942: }
943:
944: int versort(struct _info **a, struct _info **b)
945: {
946: return strverscmp((*a)->name,(*b)->name);
```

Fig. 3. A mapping produced by Conclave-Mapper: on the left the Problem Ontology can be used to constrain the concepts being searched, on the right the Program Ontology can be used to constrain the range of which program elements are being analyzed, and in the center the resulting rank sorted by relevance

The *Query* type used before describes a query supplied by the user (a predefined set of queries is also available via the system interface). A DSL was developed to describe these queries (either search or mapping); this language is discussed in the next subsection.

4.1 The Query Language

In order to compute rankings and mappings at least one query is required. This section describes the DSL that was devised to create such queries. Each query has at least three main components: (a) keywords; (b) domain and range constrains (e.g. search only functions, or variables); and, (c) the scoring function used to compute the relatedness score between the elements (some of these have default values).

The DSL query language allows two major types of queries: (a) simple strings; or (b) complex queries where a set of proprieties that define the query are supplied in the form of `PropriertyName=PropriertyValue`[9]. To distinguish simple queries from complex, the latter are enclosed in square brackets (`[ ]`). To introduce possible queries and proprieties a set of query examples are presented next.

[9] Keywords are only used for search queries by the locate function, and are ignored when creating maps.

The simplest possible query that can be written is a simple string, where the words that represent features or concepts being search are concatenated together using blank spaces, for example:

```
"color"
"color schema"
```

These queries search for elements that are closely related with the words `"color"` (or also `"schema"`). By default, a score is computed for all the Program Ontology elements (functions, variables, classes, etc.), and the scoring function used is `kpss` (scoring functions are discussed in detail in Sec. 4.2).

The program maintainer when initially addressing a possible bug fix, may be interested in searching functions (or methods) only. The `class` propriety can be used to constrain the subclass of elements that are being retrieved from the ontology. The following query performs a search for the words `"color"` and `"schema"`, but only analyses elements that are instances of the class `Function` (remember the ontology definition in Sec. 3.2, and that the Program Ontology is the default ontology for selecting elements):

```
[ word=color word=schema class=Function ]
```

This means that the elements of the resulting rank are only instances of functions, because the query constrains the search domain of the locate function. Another example, can be searching for variables, by selecting the class `Variable`, this includes all the members that are instances of the class `Variable` and also instances of all sub-classes of `Variable` (e.g. `Parameter`, `LocalVariable`), i.e. the resulting rank includes all kinds of variables in the original program:

```
[ word=color class=Variable ]
```

Particular types of variables can be selected, for example searching only local variables:

```
[ word=color class=LocalVariable ]
```

Another important propriety that can be defined is the scoring function, i.e. the function that will compute the semantic relatedness score between the keywords searched and each of the selected elements, this is done using the `score` propriety. For example, the query:

```
[ word=color class=Variable score=levenshtein ]
```

uses the `levenshtein` word distance algorithm [25], to compute the score. By default, the scoring function based on kPSS is used.

So far, the illustrated queries have been computing scores between elements (e.g. functions, variables) and a set of words. But more complex comparisons may provide more accurate rankings. The `aggr` propriety allows a query to define the name of a relation (defined in the ontology) to compute a score not only between each selected elements, but also including a set of elements that are related to them. The query:

```
[ word=color class=Function score=levenshtein aggr=inFunction ]
```

for example, analyses all the functions, and for each function also considers all the elements that are related with that function by the relation `inFunction` (defined in the ontology). This relation is used to link all the local variables and parameters to all the functions (or methods depending on programming language) where they are defined and used. In practice, the score for each element (function) is the average between computing the score for the element itself, and the score for every local variable and parameter defined in that function.

The scores used by the *locate* and *mapping* functions are calculated by the function defined in the query, and are discussed in the next section.

4.2 Scoring Functions

The score between two elements (or an element and a word) quantify how close they are semantically related. This score is used to sort the ranks computed by the *locate* function by relevance, or to highlight the cells that express close relatedness between elements in the matrixes computed with the *mapping* function.

The main scoring function available in the Conclave system is the *kpss* function (used by default), and is based on kPSS, which defines a formalism to describe synonymous sets based on Probabilist Synonymous Sets (PSS) [6, 42]. These define synonymous sets based on statistical analysis of parallel corpora.

A kPSS of order n is formally defined as a set of orders; a synonymous set corresponds to each order n (where $n \in N$), which is defined as a list of *Term*. Each *Term* contains a word and a probability:

$$\begin{aligned}&\textbf{type}\ \mathit{kPSS}\ = [\mathit{Order}]\\ &\textbf{data}\ \mathit{Order} = \mathit{Order}\ \{\ n\ ::\ \mathit{Int},\ \mathit{synsen}\ ::\ [\mathit{Term}]\ \}\\ &\textbf{data}\ \mathit{Term}\ = \mathit{Term}\ \{\ \mathit{word}\ ::\ \mathit{String},\ \mathit{prob}\ ::\ \mathit{Float}\ \}\end{aligned}$$

The sum of all the probabilities in every synonymous set, for every order is always equal to 1. A kPSS can be built for any given word, and usually only up to order 3. Every time a new order is added, probabilities need to be normalized so that the invariant is kept valid. The first order contains the original word itself only, the second order *synset* contains only words directly derived from the original word (e.g. lemma, inflection), and the third order contains terms extracted from the PSS for the same word. Table 1 illustrates the kPSS for the word "*inserting*".

Once a kPSS is available for a pair of words, the relatedness score between these words can be calculated. The *kpss* function is used to compute this score (as a *Float*) and is defined as:

$$\begin{aligned}&\mathit{kpss} &&::\ \mathit{kPSS}\ \rightarrow\ \mathit{kPSS}\ \rightarrow\ \mathit{Float}\\ &\mathit{kpss}\ k1\ k2 &&= \textstyle\sum\ [\ \mathbf{min}\ (\mathit{prob}\ x)\ (\mathit{prob}\ y)\ |\\ &&&\qquad x\ \leftarrow\ \mathit{flatten}\ k1,\ y\ \leftarrow\ \mathit{flatten}\ k2,\ \mathit{word}\ x\ ==\ \mathit{word}\ y\ ]\end{aligned}$$

This function iterates over the flattened version of the kPSS, and sums the minimum probabilities for terms that are common. The flattened version of the kPSS is simply a single list of terms. The *flatten* function is defined as:

Table 1. kPSS of order 3 for the word "*inserting*"

Order (n)	$n = 1$	$n = 2$	$n = 3$
SynSet ($word \rightarrow prob$)	$inserting \rightarrow 0,(3)$	$insert \rightarrow 0,1(6)$ $insertings \rightarrow 0,(3)$	$inserted \rightarrow 0,023$ $enter \rightarrow 0,039$ $insertion \rightarrow 0,063$ $inserts \rightarrow 0,014$ (...)

```
flatten      :: kPSS → [Term]
flatten kpss = concat [ synset order | order ← kpss ]
```

Other scoring functions can be used to produced different ranks and mappings. The *levenshtein* function is another example, this calculates the score as the word distance between terms. Another function implemented in the system is the *match* function (this helps simulating techniques based on grep[10]), that simply returns 1 if the words match, or 0 otherwise. The next section describes the experimental evaluation done to empirically verify the advantages of the use of kPSS.

5 Experimental Validation

The previous sections describe the underlying technique used in the CONCLAVE system for feature location, based on kPSS. This section describes the preliminary evaluation done, to verify if this technique introduces benefits over other common techniques. In current available IDEs, common search facilities provided to the users, are still grep-like approaches, so the following research question was formulated:

RQ1: How does the *kpss* scoring function performs, when compared to the *match* scoring function, for finding relevant elements of the code given a search query?

To help answering this question the following experience was performed:

Step 1: in order to ease the process or replicating this experience the benchmark provided by Dit *et al*[11] for the jEdit[12] editor (version 4.3) was used, instead the devising a new data set. The benchmark contains a set of 150 bug reports, including the function set that was changed to resolve the bug (refered as the gold set) – more details about the benchmark in [12];

Step 2: the title for each bug report was extracted, stop words[13] were removed, and the resulting set was archived as keywords;

[10] `http://www.gnu.org/software/grep/` (Last accessed: 29-01-2014).

[11] `http://www.cs.wm.edu/semeru/data/benchmarks/` (Last accessed: 29-01-2014).

[12] `http://www.jedit.org/` (Last accessed: 29-01-2014).

[13] Common words that tend to express poor semantics (e.g. *"the"*, *"a"*, *"too"*) [29].

Step 3: for each bug report, the *locate* function to compute a rank was called, using the *match* scoring function, the keyword set computed in *Step 2*, and setting as range the `Function` program element;
Step 4: replicate *Step 3* but using the *kpss* scoring function;
Step 5: calculate the effectiveness measure for each resulting rank.

The effectiveness measure is calculated by analyzing the computed rank in order, and its value is the first position of the rank that is a relevant function. Functions that are part of the set of functions changed to resolve the bug (the gold set) are considered relevant. The rank position can be compared for different scoring functions to measure which rank produced the best results. This approach was also used in [33] and [38] for comparing feature location techniques performance.

The results of this experience are presented in Table 2. They show that for this software package the kPSS based scoring approach produced a better result 55 times, outperforming the 22 better results achieved by the simple *match* function. The remaining times either both approaches scored the same, or none of the relevant functions were found in the resulting rank.

Table 2. Results of the experimental validation

Scoring Function	**Analysed Bugs**	**Better Eff. Measure**
match	150	22
kPSS	150	51

Although these results are satisfactory, they do not provide enough empirical data to generalize the performance of kPSS based techniques. Also, the keywords used to build the queries and the functions gold sets are a threat to validity because: (a) the keywords set was built automatically from reports titles that sometimes lack relevant terms, or use only ambiguous words (e.g. "*bug*"), a human would be more prone to devise a set of terms (after reading the report) that would create a more accurate rank; (b) sometimes, when fixing bugs, the actual defect is really not related to the concepts functions are addressing, which translates in changing code unrelated to search queries.

6 Conclusion

Many PC techniques benefit from mappings between the source code and problem domain concepts. These relations help the programmer quicker understand the source code, and discover which areas of the code need changing to address a specific feature or bug fix.

Many tools and techniques can be used to gather information about the program and the problem domain. Using ontologies allows the combination of heterogenous results and data in a single representation format. Applications can take advantage of a panoply of tools available (e.g. inference engines, descriptive logics, OTK-*like* frameworks), to perform data analysis and relate elements in different domains. kPSS based feature location is a sound example of such applications. The OTK framework for abstracting ontology operations from the underlying technology has proven a valuable asset during applications implementation.

The main trends for future work include devising new scoring functions, as well as combinations of approaches to improve results. And also, design practical experiments to compare Conclave-Mapper with other state-of-the-art feature location techniques.

References

1. Antoniol, G., Guéhéneuc, Y.-G.: Feature identification: An epidemiological metaphor. IEEE Transactions on Software Engineering 32(9), 627–641 (2006)
2. Bechhofer, S., Van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A., et al.: Owl web ontology language reference. W3C Recommendation 10, 2006–01 (2004)
3. Biggerstaff, T.J., Mitbander, B.G., Webster, D.: The concept assignment problem in program understanding. In: Proceedings of the 15th International Conference on Software Engineering, pp. 482–498. IEEE Computer Society Press (1994)
4. Binkley, D., Lawrie, D.: Information retrieval applications in software maintenance and evolution. In: Encyclopedia of Software Engineering (2009)
5. Binkley, D., Lawrie, D.: Information retrieval applications in software development. In: Encyclopedia of Software Engineering (2010)
6. Carvalho, N.R., Almeida, J.J., Pereira, M.J.V., Henriques, P.R.: Probabilistic synset based concept location. In: SLATE 2012 — Symposium on Languages, Applications and Technologies (June 2012)
7. Chen, K., Rajlich, V.: Case study of feature location using dependence graph. In: 8th International Workshop on Program Comprehension. IEEE (2000)
8. Chikofsky, E.J., Cross II, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Software, 13–17 (1990)
9. Corbi, T.A.: Program understanding: Challenge for the 1990s. IBM Systems Journal 28(2), 294–306 (1989)
10. Deissenboeck, F., Pizka, M.: Concise and consistent naming. Software Quality Journal 14(3), 261–282 (2006)
11. Dit, B., Guerrouj, L., Poshyvanyk, D., Antoniol, G.: Can better identifier splitting techniques help feature location? In: IEEE 19th International Conference on Program Comprehension (2011)
12. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process 25(1), 53–95 (2013)
13. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Transactions on Software Engineering 29(3), 210–224 (2003)
14. Furnas, G.W., Landauer, T.K., Gomez, L.M., Dumais, S.T.: The vocabulary problem in human-system communication. Communications of the ACM 30(11), 964–971 (1987)

15. Happel, H.-J., Seedorf, S.: Applications of ontologies in software engineering. In: Proc. of Workshop on Sematic Web Enabled Software Engineering (SWESE) on the ISWC, pp. 5–9. Citeseer (2006)
16. Hayashi, S., Yoshikawa, T., Saeki, M.: Sentence-to-code traceability recovery with domain ontologies. In: 2010 17th Asia Pacific Software Engineering Conference (APSEC), pp. 385–394. IEEE (2010)
17. Hill, E., Pollock, L., Vijay-Shanker, K.: Exploring the neighborhood with dora to expedite software maintenance. In: Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 14–23 (2007)
18. Hill, E., Pollock, L., Vijay-Shanker, K.: Automatically capturing source code context of nl-queries for software maintenance and reuse. In: Proceedings of the 31st International Conference on Software Engineering. IEEE (2009)
19. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: the making of a Web Ontology Language. Web Semantics: Science, Services and Agents on the World Wide Web 1(1), 7–26 (2003)
20. Keller, W.: Mapping objects to tables. In: Proc. of European Conference on Pattern Languages of Programming and Computing, Kloster Irsee, Germany, vol. 206, p. 207. Citeseer (1997)
21. Klyne, G., Carroll, J.J., McBride, B.: Resource description framework (rdf): Concepts and abstract syntax. W3C Recommendation, 10 (2004)
22. Lattner, C.: Llvm and clang: Next generation compiler technology. In: The BSD Conference, pp. 1–2 (2008)
23. Lawrie, D., Binkley, D.: Expanding identifiers to normalize source code vocabulary. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 113–122 (2011)
24. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: What's in a name? a study of identifiers. In: 14th International Conference on Program Comprehension (2006)
25. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 707–710 (1966)
26. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An information retrieval approach to concept location in source code. In: Proceedings of the 11th Working Conference on Reverse Engineering, pp. 214–223. IEEE (2004)
27. Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., Sergeyev, A.: Static techniques for concept location in object-oriented code. In: Proceedings of the 13th International Workshop on Program Comprehension, IWPC 2005, pp. 33–42. IEEE (2005)
28. Marcus, A., Rajlich, V.: Identification of concepts, features, and concerns in source code. In: Panel Discussion at the International Conference on Software Maintenance (2005)
29. Martin, J.H., Jurafsky, D.: Speech and language processing (2000)
30. Nelson, M.L.: A survey of reverse engineering and program comprehension. Arxiv preprint cs/0503068 (2005)
31. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf (2013)
32. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
33. Poshyvanyk, D., Guéhéneuc, Y.-G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Transactions on Software Engineering 33(6), 420–432 (2007)

34. Prud'Hommeaux, E., Seaborne, A., et al.: Sparql query language for rdf. W3C Recommendation, 15 (2008)
35. Rajlich, V., Wilde, N.: The role of concepts in program comprehension. In: Proceedings of the 10th International Workshop on Program Comprehension, pp. 271–278. IEEE (2002)
36. Ratiu, D., Deissenboeck, F.: How programs represent reality (and how they don't). In: 13th Working Conference on Reverse Engineering, WCRE 2006, pp. 83–92. IEEE (2006)
37. Ratiu, D., Deissenboeck, F.: From reality to programs and (not quite) back again. In: 15th IEEE International Conference on Program Comprehension, ICPC 2007, pp. 91–102. IEEE (2007)
38. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: 2010 IEEE 18th International Conference on Program Comprehension (ICPC), pp. 14–23. IEEE (2010)
39. Robillard, M.P.: Topology analysis of software dependencies. ACM Transactions on Software Engineering and Methodology (TOSEM) 17(4), 18 (2008)
40. Safyallah, H., Sartipi, K.: Dynamic analysis of software systems using execution pattern mining. In: 14th IEEE International Conference on Program Comprehension (2006)
41. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: Proceedings of the 6th International Conference on Aspect-Oriented Software Development, pp. 212–224. ACM (2007)
42. Simões, A., Almeida, J.J., Carvalho, N.R.: Defining a probabilistic translation dictionaries algebra. In: XVI Portuguese Conference on Artificial Inteligence - EPIA, pp. 444–455 (September 2013)
43. Von Mayrhauser, A., Vans, A.M.: Program comprehension during software maintenance and evolution. Computer 28(8), 44–55 (1995)
44. Wilde, N., Buckellew, M., Page, H., Rajlich, V., Pounds, L.: A comparison of methods for locating features in legacy software. Journal of Systems and Software (2003)
45. Würsch, M., Ghezzi, G., Reif, G., Gall, H.C.: Supporting developers with natural language queries. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1 (2010)
46. Zhang, Y.: An Ontology-based Program Comprehension Model. PhD thesis (2007)
47. Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F.: Sniafl: Towards a static noninteractive approach to feature location. ACM Trans. Softw. Eng. Methodol. 15(2), 195–226 (2006)