Conclave: Writing Programs to Understand Programs

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

1 **Departamento de Informática/CCTC**
Universidade do Minho, Braga, Portugal
{narcarvalho,jj,prh}@di.uminho.pt
2 **Escola de Tecnologia e Gestão/CCTC**
Instituto Politécnico de 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.

This paper introduces Conclave, an environment for software analysis, that enhances program comprehension activities. Programmers use natural languages to describe and discuss the problem domain, programming languages to write source code, and markup languages to have programs talking with other programs, and so this system has to cope with this heterogeneity of dialects, and provide tools in all these areas to effectively contribute to the understanding process. The source code, the problem domain, and the side effects of running the program are represented in the system using ontologies. A combination of tools (specialized in different kinds of languages) create mappings between the different domains. Conclave provides facilities for feature location, code search, and views of the software that ease the process of understanding the code, devising changes. The underlying feature location technique explores natural language terms used in programs (e.g. function and variable names); 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 respectively.

1998 ACM Subject Classification D.2.7 Distribution, Maintenance, and Enhancement - Restructuring, reverse engineering, and reengineering

Keywords and phrases software maintenance; software evolution; program comprehension; feature location; concept location; natural language processing

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.0

1 Introduction

Reality shifts, bug fixes, updates or introduction of 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 [41]. This task is probably the main challenge during software maintenance activities [10]. The

3rd Symposium on Languages Technologies and Applications (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal and Alberto Simões; pp. 0–15

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

programmer is able to understand the program when he or she can explain the source code, and relate the code with the concepts in its problem domain [3].

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. Many of the techniques used in reverse engineering rely on mappings between human oriented concepts (described using natural language), and program elements (implemented using programming languages) [33]. These are often used to locate which parts of the program are responsible for addressing specific domain concepts [3], and are usually referred in the literature as feature location techniques [11].

Natural languages are used to describe and discuss real world problems, and programming languages are used to develop computer programs that address these problems. Although, programming languages have unambiguous grammars and limit the sentences that can be used to write software, still give some degree of freedom 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 and possible relations between different language domains are typically described as textual analysis, often combined with static analysis [11].

This paper introduces Conclave[2], a system of tools for software analysis. 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. The system provides a set of facilities for searching and a feature location technique, that measures semantic relatedness between source code elements, and elements supplied by the maintainer as query searches. Several views provide mappings between source code and real world concepts, facilitating feature location activities. The underlying feature location technique 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[3] between them. The main output of this tool is a list of ranks – sorted by relevance – of program elements that are prone to address some specific real world domain concept. The system also provides a Domain Specific Language (DSL), for writing search queries.

The next section introduces the Conclave system, including a brief description about the major stages of the system workflow. Section 3 describes in more detail some tools and results that can be produced using the system. Section 4 presents related work, and introduces some state-of-the-art techniques for feature location. Section 5 describes the experimental validation held to do a preliminary evaluation of some Conclave tools and

[1] Available from: `http://www.python.org/dev/peps/pep-0008/` (Last accessed: 29-01-2014).
[2] Conclave website: `http://conclave.di.uminho.pt` (Last accessed: 10-03-2014).
[3] 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.

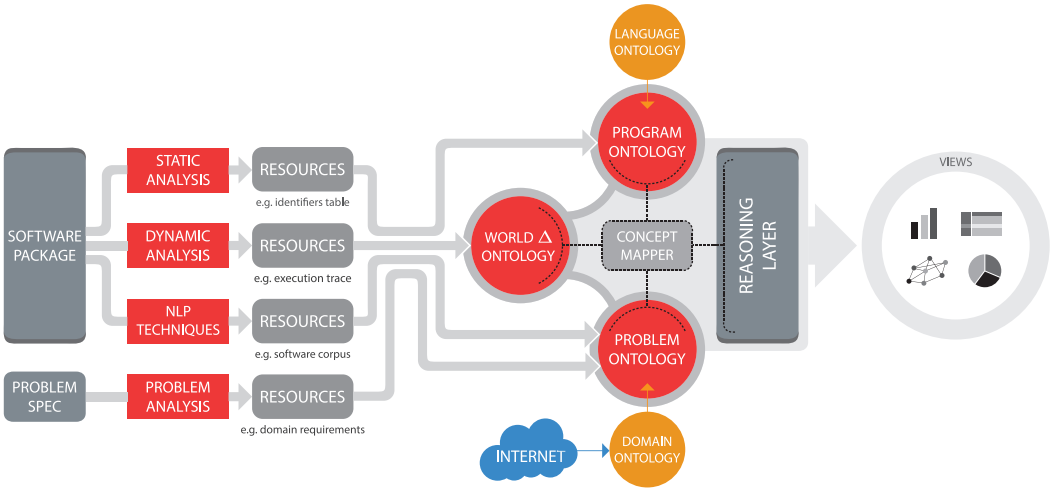

Figure 1 Overview of the major stages of the CONCLAVE system workflow.

techniques, including results discussion. Finally, Section 6 presents some final remarks and trends for future work.

2 CONCLAVE Architecture Overview

The CONCLAVE environment provides a set of tools to perform software analysis. The main system workflow is divided in three 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 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 other workflows.

2.1 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), Natural Language Processing (NLP) approaches (e.g. processing non-source code content for domain vocabulary), 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 [29], for gathering program identifiers information for `Java` programs;

DMOSS: is a toolkit for software documentation assessment. It produces an attribute tree representation of a software package, and other software related resources like the *documentation corpus* that is used later to create a initial version of the problem ontology. For more details about this framework refer to [8].

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 resources, all the 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.

2.2 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 problem domain;

World △ 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 allows 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 [30, 32].

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)[4] was developed, 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.

[4] Implemented as a set of libraries for the Perl programming language.

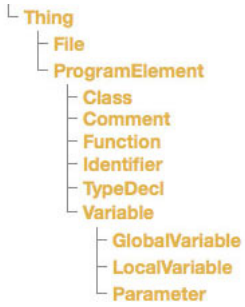

Figure 2 Program ontology sub-set of the class hierarchy.

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 a set of lines that look something like[5]:

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

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 the string '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 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[6]. 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. [35,43,44]). This also eases future integration processes with other tools that followed similar 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 information gathered from different elements and domains to build semantic bridges between elements. More details about this stage are discussed in the next section.

2.3 Reasoning and Views

During this stage more knowledge about the system is built 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

[5] More examples available in the tool website: `http://conclave.di.uminho.pt/clang` (Last accessed: 27-01-2014).

[6] 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].

Table 1 Some AbcMidi package characteristics.

Total Files	Size (KLOC[9])	Total Ids.	Multi-word Ids.
86	∼ 33	3437	2142 (62%)

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 case the output is a view about a particular aspect of 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, one of the tools described in the next section, is an example of tools that are used during this stage.

3 Conclave Quick Tour

The goal of this section is to illustrate some practical applications of the Conclave system. Two tools are introduced, and some features are illustrated. The software analyzed and used in the next examples in this section is AbcMidi (version 2012.12.25)[7], a package that provides a set of tools to convert Abc[8] files to the Midi format. Table 1 presents some characteristics about this software package.

Figure 3 illustrates the Conclave web interface front page, the system is divided in blocks, and most of the applications use resources produced by other blocks. The tools presented in this section address two popular problems in the context of program comprehension: (1) splitting multi-term program identifiers, and (2) mappings between program elements and real world concepts.

3.1 Splitting Identifiers: Lingua-IdSplitter

Lingua-IdSplitter (henceforth abbreviated LIdS) is a simple and fast algorithm that addresses the problem of splitting *soft words*[10] that compose an identifier. It handles abbreviations, acronyms, or any type of linguistic short-cuts (for example, use only the first letter of a word). The algorithm calculates a ranked list of all the possible splits for an identifier, based on a set of dictionaries, and the top entry in the rank is proposed as the correct split.

[7] Available from `http://abc.sourceforge.net/abcMIDI/` (Last accessed: 11-03-2014).
[8] A text notation to represent music.
[9] Thousands Lines of Code.
[10] Usually refers to words that are combined together to create an identifier without using an explicit mark between them (e.g., *"timesort"*) [24].

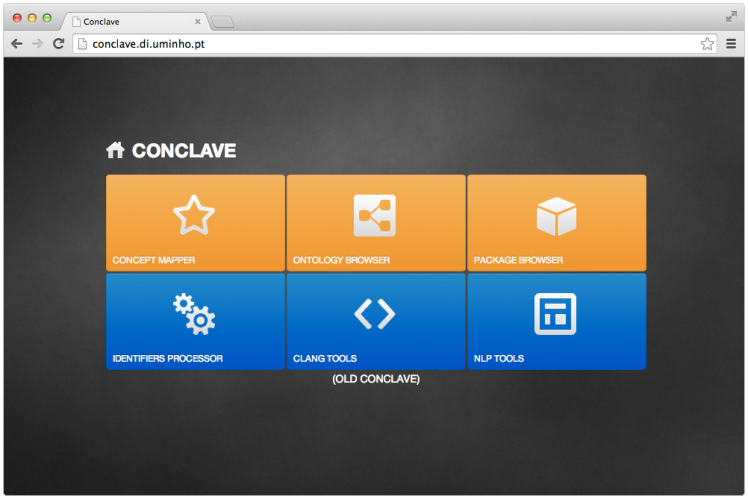

Figure 3 CONCLAVE system web interface front page, main applications are divided in blocks.

Besides the actual split, the result includes the set of full terms that compose the identifier, in case abbreviations were used for example. This technique can use an arbitrary set of dictionaries, but one of the benefits introduced by this approach is the use of a software specific dictionary computed automatically from a software package corpus – also computed automatically and specific to each software package – using a combination of Natural Language Processing (NLP) techniques. This dictionary enables the algorithm to correctly handle identifiers splitting using arbitrary abbreviations or combinations of term specific to the application domain, not prone to be present in more general programming dictionaries. this technique can also cope with identifiers that use explicit marks, like underscores (e.g., *"time_sort"*) or the *CamelCase* notation (e.g., *"timeSort"*).

This tool is implemented as a Perl library that can be used in other tools and contexts, and is available for download in the official Perl library archive[11]. In the context of PC this is relevant when dealing with program identifiers (e.g., variable names, function names) that were created using a combination of abbreviation and words. Correctly splitting program identifiers has a direct impact on future programming comprehension techniques [24]. Even a simple program can have thousands of identifiers, undertake this task manually would be unfeasible, so the literature is rich on techniques to address this problem (e.g., [13, 14, 23]).

In the CONCLAVE system this tool retrieves the identifiers from the ontology, making it independent of the programming language used. The resulting split and expansion sets are loaded to the ontology and related to each identifier, so they are readily available for other applications to use. Table 2 illustrates the split and term sets computed for some ABCMIDI identifiers, and Figure 4 is a screenshot of CONCLAVE identifiers table for ABCMIDI including the splits and terms sets computed by LIDS.

LIDS algorithm for computing *soft splits* starts by computing all the possible valid strings that can be found starting at every position of the identifier. A string is considered valid if it is successfully found in any of the dictionaries being used. The next step is to build an automaton, with all the strings found, to calculate all the possible sequence of nodes (paths), that concatenate to rebuild the original identifier. The set of paths in the automaton defines

[11] Available from: `http://search.cpan.org/dist/Lingua-IdSplitter/` (Last accessed: 18-03-2014).

Table 2 AbcMidi identifiers examples, and corresponding splits and abbreviation expansions.

Identifier	Splits	Expands
mrest	*m* | *rest*	{ *multibar*, *rest* }
timesig	*time* | *sig*	{ *time*, *signature* }
chan	*chan*	{ *channel* }

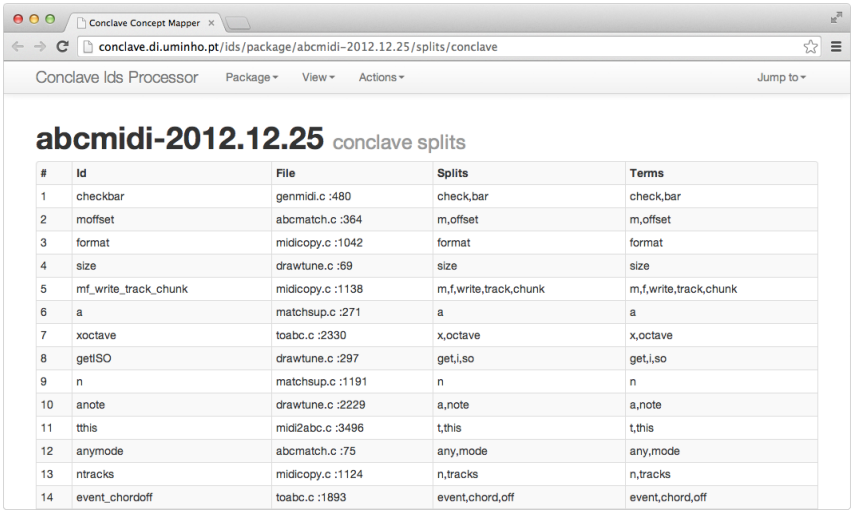

Figure 4 Conclave interface to view the splits and terms set for all identifiers in the package being analyzed.

the set of string sequences that are candidates to be the identifier correct split. Next, the algorithm computes the score for each candidate, creating a rank, where the top element (the sequence with the higher score) is the resulting split.

The formula to calculate the score for a given sequence is analytically defined as:

$$score(S) = \frac{(\prod_{i=1}^{length(S)} factor(S_i)) + length(m)}{length(S)^2}$$

where the multiplicand of factors (a factor is calculated for each element in the sequence) plus the length of the longer string in the sequence, is normalized by the squared sequence length. Each factor is calculated according to the formula:

$$factor(s, t, w) = length(s) \times w$$

i.e., the length of the string found times the dictionary weight that validated the string.

The final result sets of terms are loaded to the program ontology, and related with the corresponding identifier. These sets of terms, can then be used to compute relatedness with words from other domains by other applications, like the one described in the next section. More details about this technique in [7].

3.2 Creating Mappings: Conclave-Mapper

Conclave-Mapper is an application that relies on data computed by other tools (see Sec. 2.1 and 2.2), 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 approaches can be used to find which parts of the code are responsible for implementing a domain concept – feature location.

A rank is defined as a collection of entries, where each entry contains the semantic relatedness score, between the element and the search query. An 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.), so elements in different ontologies can be related. A map is defined 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.

The application implements two main functions to compute each one of the available output types. The *locate* function creates a rank and has the following signature:

$$locate :: Query \rightarrow Rank$$

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. The *mapping* function creates a map and has the following signature:

$$mapping :: Query \rightarrow Query \rightarrow ScoreFunction \rightarrow Map$$

This function, given two queries, and a scoring function, 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 5 illustrates a possible view of these mappings, highlighting the best relevance ranking between the application domain and functions.

The *Query* type used before describes a query supplied by the user (a pre-defined set of queries is also available via the system interface). A DSL was developed to describe these queries (either search or mapping). 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 (all except keywords have default values). To illustrate the DSL some query examples are given below.

The following query performs a search for the words `"color"` and `"schema"`, but only analyses elements that are instances of the class `Function`.

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

The next query searches variables for the word `"color"`, and uses the `levenshtein` word distance algorithm [25], to compute the score. By default, the scoring function based on kPSS is used.

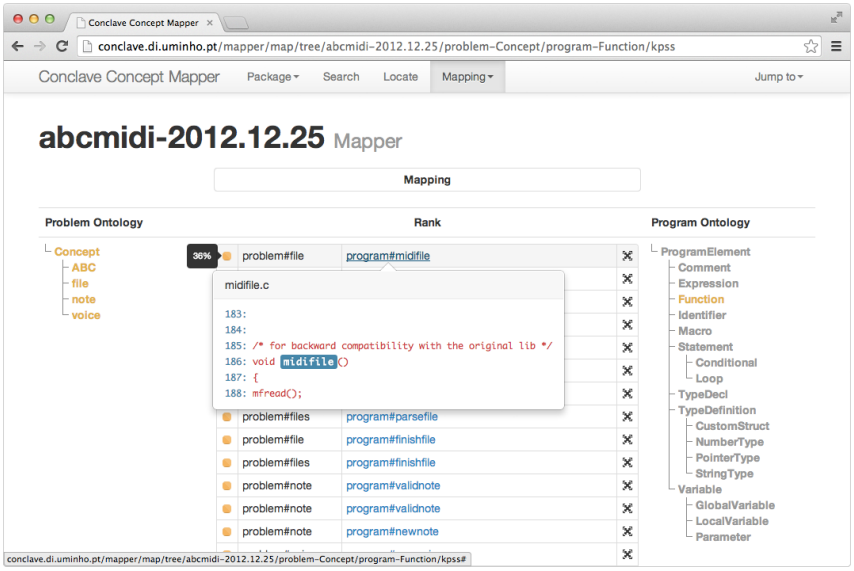

Figure 5 A mapping produced by CONCLAVE-MAPPER: on the left the Problem Ontology can be used to constrain the concepts being searched, on the right the Program Ontology can be used to constrain the range of which program elements are being analyzed, and in the center the resulting rank sorted by relevance (hovering the program element shows the corresponding zone in the file where the element appears).

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

The following query, searches 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):

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

The `inFunction` 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 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) [5, 40]. These define synonymous sets based on statistical analysis of parallel corpora.

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:

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

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 and corresponding probabilities.

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[12]), that simply returns 1 if the words match, or 0 otherwise. Full details about CONCLAVE-MAPPER available in [6].

4 Related Work

Program Comprehension (PC) is a field of research concerned with devising ways to help programmers understand software systems. In this context, 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, 33].

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, 38]); (b) static analysis, based on static source code information, such as slicing, control or data flow graphs (e.g. [9, 27, 37]); 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, 26]), NLP (e.g. [18, 39]), or pattern matching (sometimes also referred as grep-*like*) based approaches (e.g. [12]). For more details about different trends and other approaches please refer to surveys [11] and [42].

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 [45], 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 [34], 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]).

5 Experimental Validations

This section describes two evaluations done for the tools illustrated in Section 3.

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

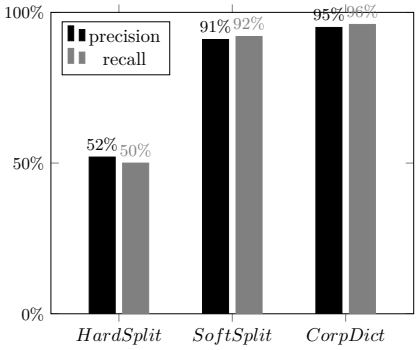

Figure 6 Precision and recall means for correct splits.

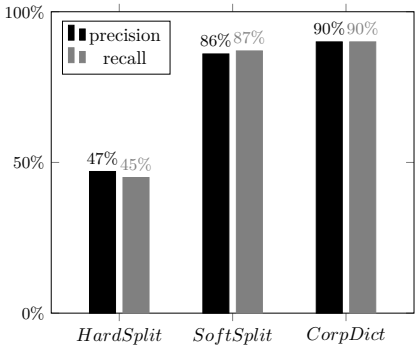

Figure 7 Precision and recall means for correct terms.

5.1 Splitting Evaluation

Section 3.1 describes the identifier splitting technique available in Conclave. This section briefly describes the experimental study undertaken to evaluate the technique ability to correctly split and expand multi-term identifiers. The following research questions were defined:

RQ1: What is the percentage of identifiers in a program that LIdS can correctly split?

RQ2: What is the percentage of identifiers in a program that LIdS can correctly split and expand in case abbreviations were used?

To help answering these questions the following experience was performed:

Step 1: Create the *oracle*, i.e., for every AbcMidi identifier manually create the correct split set, and correct term set. (this was done by the authors, and in cases where the was not an agreement, or the original programmer purpose was not clear, the identifier was not included).

Step 2: Compute the split and terms sets for every identifier in the *oracle* using LIdS hard-split function.

Step 3: Compute the split and terms sets for every identifier in the *oracle* using LIdS soft-split function, providing general purpose dictionaries.

Step 4: Compute the split and terms sets for every identifier in the *oracle* using LIdS soft-split function, providing general purpose dictionaries and the software specific dictionary.

Step 5: Compare sets computed in *Step 2-4* and the sets manually created in *Step 1* and measure precision and recall.

For a given identifier *id* to split let the *oracle* split set be: $o = \{o_1, o_2, ..., o_n\}$, and $s = \{s_1, s_2, ..., s_n\}$ the computed split, then the precision and recall are calculated as:

$$precision = \frac{|o \cap s|}{|s|} \quad recall = \frac{|o \cap s|}{|o|}$$

where $|x|$ represents the cardinality of x. The same formulae are applied when calculating the measures for correct terms, but using the calculated sets of terms instead of splits.

Figures 6 and 7 illustrate the measurement results. For this software package the proposed technique was able to correctly split and expand almost all identifiers (precision and recall in the order of 90%). More details about this evaluation and comparisons with other techniques in [7].

Table 3 Better effective measure for different approaches for the jEdit benchmark.

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

5.2 Query Search Evaluation

Section 3.2 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 available to programmers, 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*[13] for the jEdit[14] 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 (referred as the gold set) – more details about the benchmark in [11];

Step 2: the title for each bug report was extracted, stop words[15] were removed, and the resulting set was archived as keywords;

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 [31] and [36] for comparing feature location techniques performance.

The results of this experience are presented in Table 3. They show that for this software package the kPSS based scoring approach produced a better result 51 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.

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

[13] Available from: `http://www.cs.wm.edu/semeru/data/benchmarks/` (Last accessed: 29-01-2014).
[14] Available from: `http://www.jedit.org/` (Last accessed: 29-01-2014).
[15] Common words that tend to express poor semantics (e.g. *"the"*, *"a"*, *"too"*) [28].

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. Full details about this experiment and other case studies in [7].

6 Conclusion

Systems like CONCLAVE enables software engineers to devise mappings between the source code and problem domain concepts. These relations help the programmer to understand quicker the software, 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 programs and the problem domain. The quicker the information is abstracted, the quicker other applications can use it. Using ontologies allows the combination of heterogenous results and data in a single representation format. Also, 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 functions to score relations between elements in the different ontologies, as well as combinations of approaches to produce more resources, and to convey more semantic information to ontologies with current available resources.

Acknowledgements This work is funded by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

References

1 Giuliano Antoniol and Y-G Guéhéneuc. Feature identification: An epidemiological metaphor. *Software Engineering, IEEE Transactions on*, 32(9):627–641, 2006.

2 Sean Bechhofer, Frank Van Harmelen, et al. Owl web ontology language reference. *W3C recommendation*, 10:2006–01, 2004.

3 T.J. Biggerstaff, B.G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1994.

4 David Binkley and Dawn Lawrie. Information retrieval applications in software development. *Encyclopedia of Software Engineering*, 2010.

5 Nuno Ramos Carvalho, José João Almeida, Maria João Varanda Pereira, and Pedro Rangel Henriques. Probabilistic synset based concept location. In *SLATE'12 — Symposium on Languages, Applications and Technologies*, June 2012.

6 Nuno Ramos Carvalho, José João Almeida, Maria João Varanda Pereira, and Pedro Rangel Henriques. Conclave: Ontology-driven measurement of semantic relatedness between source code elements and problem domain concepts. In *14th International Conference on Computational Science and Its Applications (ICCSA)*, 2014. [forthcoming].

7 Nuno Ramos Carvalho, José João Almeida, Maria João Varanda Pereira, and Pedro Rangel Henriques. From source code identifiers to natural language terms. *Journal of Systems and Software*, 2014. [under review process].

8 Nuno Ramos Carvalho, Alberto Simões, and José João Almeida. Open source software documentation mining for quality assessment. In *Advances in Information Systems and Technologies*, volume 206 of *Advances in Intelligent Systems and Computing*, pages 785–794. Springer Berlin Heidelberg, 2013.

9 Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *8th International Workshop on Program Comprehension*. IEEE, 2000.

10 T.A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

11 Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 2013.

12 George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

13 Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

14 Latifa Guerrouj, Philippe Galinier, Y Gueheneuc, Giuliano Antoniol, and Massimiliano Di Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012.

15 Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pages 5–9. Citeseer, 2006.

16 Shinpei Hayashi, Takashi Yoshikawa, and Motoshi Saeki. Sentence-to-code traceability recovery with domain ontologies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 385–394. IEEE, 2010.

17 Emily Hill, Lori Pollock, and K Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of 22nd IEEE/ACM international conference on Automated software engineering*, pages 14–23, 2007.

18 Emily Hill, Lori Pollock, and K Vijay-Shanker. 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 I. Horrocks, P.F. Patel-Schneider, and F. van Harmelen. 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 Wolfgang Keller. Mapping objects to tables. In *Proc. of European Conference on Pattern Languages of Programming and Computing, Kloster Irsee, Germany*. Citeseer, 1997.

21 Graham Klyne, Jeremy J Carroll, and Brian McBride. Resource description framework (rdf): Concepts and abstract syntax. *W3C recommendation*, 10, 2004.

22 Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

23 D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *27th IEEE International Conference on Software Maintenance*, pages 113–122, 2011.

24 D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th International Conference on Program Comprehension*, 2006.

25 Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

26 A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

27 Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 33–42. IEEE, 2005.

28 James H Martin and D Jurafsky. Speech and language processing, 2000.

29 Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

30 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *The Semantic Web-ISWC 2006*, pages 30–43. Springer, 2006.

31 Denys Poshyvanyk, Y-G Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 2007.

32 Eric Prud'Hommeaux, Andy Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

33 V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. IEEE, 2002.

34 Daniel Ratiu and Florian Deissenboeck. How programs represent reality (and how they don't). In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 83–92. IEEE, 2006.

35 Daniel Ratiu and Florian Deissenboeck. From reality to programs and (not quite) back again. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 91–102. IEEE, 2007.

36 Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 14–23. IEEE, 2010.

37 Martin P Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(4):18, 2008.

38 H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *14th IEEE International Conference on Program Comprehension*, 2006.

39 David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM, 2007.

40 Alberto Simões, José João Almeida, and Nuno Ramos Carvalho. Defining a probabilistic translation dictionaries algebra. In *XVI Portuguese Conference on Artificial Inteligence - EPIA*, pages 444–455, September 2013.

41 A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

42 Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 2003.

43 Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C Gall. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010.

44 Yonggang Zhang. *An Ontology-based Program Comprehension Model*. PhD thesis, Concordia University, Montreal, Quebec, Canada, 2007.

45 Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, April 2006.