Probabilistic *SynSet* Based Concept Location

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

1 **Departamento de Informática, Universidade do Minho**
Braga, Portugal
{narcarvalho,jj,prh}@di.uminho.pt

2 **Escola Superior de Tecnologia e Gestão, Instituto Politécnico de Bragança**
Bragança, Portugal
mjoao@ipb.pt

Abstract

Concept location is a common task in program comprehension techniques, essential in many approaches used for software care and software evolution. An important goal of this process is to discover a mapping between source code and human oriented concepts.

Although programs are written in a strict and formal language, natural language terms and sentences like identifiers (variables or functions names), constant strings or comments, can still be found embedded in programs. Using terminology concepts and natural language processing techniques these terms can be exploited to discover clues about which real world concepts source code is addressing.

This work extends symbol tables build by compilers with ontology driven constructs, extends synonym sets defined by linguistics, with automatically created Probabilistic *SynSets* from software domain parallel corpora. And using a relational algebra, creates semantic bridges between program elements and human oriented concepts, to enhance concept location tasks.

1998 ACM Subject Classification D.2.5 Testing and Debugging: code inspections and walkthroughs

Keywords and phrases program comprehension, program visualization, concept location, code inspection, synonym sets, probabilistic synonym sets, translation dictionary

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.239

1 Introduction

Program comprehension provides valuable insight in many software evolution and maintenance tasks: bug hunting, fixing, feature improvements, etc. Reverse engineering techniques often rely on a mapping between human oriented concepts and program elements [15]. This mapping is required mainly because there is a big gap between the natural languages used to discuss and describe concepts in the problem domain, and the formal programming languages used to actually implement them [5]. To address this issue a clear definition of the program elements is required and also a way to relate these elements with human oriented concepts.

Although programming languages grammars are very formal and strict, and strongly limit the expressions and statements that can be composed to write programs, some small degree of freedom is still given to the programmer to use some more natural terms, when writing comments or naming functions and variables for example. These terms can give clues to which concepts the implementation is addressing, and the meaningfulness of these terms can have a direct impact in future program comprehension approaches [14].

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 239–253

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

In order to close this gap between languages a new structure needs to be devised to represent program elements with information derived and extracted from program identifiers. This work introduces the use of a Ontology Oriented Symbol Table (*OntOSymbolTable*), to represent elements in the program. A symbol table enriched with relations gathered from static analysis performed on source code. This table is used to represent some of the elements in the program, which gives a more structured and formal way to reason about source code. Program identifiers are present in this table and directly related with them is a data structure called Probabilistic Synonym Set (*ProbSynSet*), an extended version of synonymous sets used by linguistics to gather terms that are conceptually equivalent. These key structures are used as clamps to sustain conceptual bridges between program and other elements and are built from a terminology translation memory (in simple terms a dictionary) called *PTD*. This particular dictionary is built from parallel corpora using Natural Language Processing techniques [19], and can be used to expand the terms used to represent a concept. An important detail is that the text used to build the parallel corpora constrains the domain of terms that appear in the final dictionaries, this helps in keeping the calculated related terms in the same context. A complete definition and more details about these structures is presented in section 3.

The next section discusses other work in this area, and uses some of this work to motivate and substantiate the approach introduced in this paper. Section 3 introduces some concepts and definitions. Section 4 describes the relational algebra devised to relate program elements with other concepts, and Section 5 discusses how this algebra can be used in the context of concept location. Section 6 illustrates some case studies and experimental validation done so far to support the initial claims. And the last section concludes with some final notes and future work.

2 Related Work

Previous work shows the relevance of program identifiers when reverse engineering programs. *Lawrie et al* have shown that terms used as program identifiers have a direct impact on future comprehension tasks quality and accuracy [14]. A study by *Takang et al*, also shows that programs that use full terms for program identifiers, instead of abbreviations, are easier to understand [20].

Caprile et al state that program identifiers are one of the most relevant source of information about programs. This was so important that their work was about restructuring named identifiers to improve other program comprehension activities [6].

The relevance of program identifiers used clearly affects future program comprehension tasks, but how much can we rely on this source of information? The work of *Anquetil et al* try do define what it means to have a "reliable naming convention" [3] to later improve program readability, *Deissenboeck et al* propose a formal model that provides rules for concise and consistent naming [7].

Abebe et al presented their use of Natural Language Processing techniques for parsing program to extract concepts [1]. In this work they build an ontology from domain concepts extracted from source code, that can be later used to suggest which files can be more relevant to a specific software change. Although, there are many similar facts between this approach and the one described in this paper, one major difference is that natural language resources used by *Abebe et al* are found in the program, and some algorithms presented in this paper take advantage of resources built outside the program scope. Also the suggested elements where the concept can be found is more accurate than a file.

Falleri et al also used Natural Language Processing techniques to enhance the extraction of concepts from program identifiers analysis [10]. Their extracted artifact is similar to a WordNet [11] and can be used later to browse concepts found in the program in a hierarchical structure. Although, their work shares many objectives with this work, they are actually quite different, mainly because *Falleri et al* are solely based their lexical structure in terms used as identifiers, while in this work identifiers terms are expanded, and bags of conceptually equivalent words are used.

Lawrie et al also discussed how to expand possible abbreviations found in term used as program identifiers [13]. This work is directly related with the one described in this paper, and furthermore an inclusion on this expanding approach into this workflow would surely benefit the final results. Since the final terms used to map concepts would be using more meaningfully vocabulary.

In other work, *Enslen et al* introduced an algorithm to automatically split program identifiers in sequences of words. This is required mainly because unlike natural languages, identifiers do not use spaces or punctuation to join lists of terms (due to programming language syntax constrains), so other techniques are used [9]. Introducing this algorithm in the work described in this paper would probably help produce better results, since instead of processing an identifier that contains more than one term, for example using *CamelCase*, the list of terms could be processed for more accurate results.

The approaches described in this paper could also be used to complement other existent work. For example, *Bacchelli et al* discussed how to link e-mails free text with software artifacts [4]. The approach described in this paper could be used to help mapping concepts from both contexts.

These are some examples of previous work that help and motivate for improving techniques and approaches for exploring program identifiers found in source code [8]. And also show that there is a direct link between the meaningfulness of terms used as program identifiers and the degree of confidence and accuracy of the mapping between source code and human concepts build based on those identifiers. The accuracy of this mapping can improve future reverse engineering tasks, and improve program comprehension techniques and results.

3 Definitions and Concepts

To make it easier to discuss our contributions, this section presents some concepts and definitions.

3.1 Ontology Oriented Symbol Table

A symbol table is a data structure used to hold information about source code constructs, usually created by a compiler or interpreter [2]. Entries in the symbol table contain information about program identifiers, such as type or scope (it can vary depending on the language transformation being done). In its' simple form a symbol table can be defined as (note that some important data is being omitted, memory addresses for variables for example, this definition emphasizes the information required in the context of this work), that is why this structure is called $PseudoSymbolTable$. Another advantage of this simplification is that most probably it won't be necessary a full-featured compiler to build this table, more practical details on this subject in Section 4. In summary this table is a list, one triple for

each identifier found in the program:

$$PseudoSymbolTable = (id \times type \times scope)*$$

where,
id is the string that represents the term used as identifier;
type is the type of identifier (variable, function, etc.);
scope the scope where the identifier is declared, global or local, or a specific code block, sometimes this could be simply the line where the identifier appears.

This section introduces an Ontology Oriented Symbol Table ($OntOSymbolTable$) definition that will be used later to create the resources required to implement concept location. This table is defined as an hash-map [1] where the keys are program constructs identifiers ($PCid$) and the values are defined by the $Entry$ datatype:

$$OntOSymbolTable = PCid \rightharpoonup Entry$$

$$\begin{array}{rcrcl} Entry & = & id & : & String \times \\ & & type & : & PCType \times \\ & & prt & : & PCid \times \\ & & src & : & (File \times Line) \times \\ & & pss & : & ProbSynSet \\ & & & & (...) \end{array}$$

For each program identifier found a new entry is created in the table, where:
PCid is a string identifying a program construct, this is unique for an entire program (even for programs written across multiple files);
id is a string that represents the actual term used as identifier;
type the identifier type, not the type of variable the identifier is declaring (defined types are described later), when in a ontology context this represents the IS_A relation;
prt represents the parent for this program element, for example a function, code block, object or method, the element is identified by its' corresponding $PCid$, in an ontology context this represents the IN_CTX relation;
src specifies where the identifier can be found in the persistent storage medium, typically a filename and a line number;
pss the $ProbSynSet$ calculated from the identifier id, more details on this data structure in the next section;
(...) states that this definition is not complete, more information is to be added like call graphs edges and dependencies, but these elements are the most relevant in this article's scope.

The following types are defined to use has $PCType$: m

$$PCType = P + V + K + M$$

where,
P represents a procedure, can be used to represent functions or methods;
V represents a variable, used to represent local and global variables;

[1] An association between keys and values, where the keys used are the program unique identifiers, and the values the corresponding calculated structure for the identifier.

```
1 P(authenticate)@F(auth.c:14)
2 V(password)@F(auth.c:3)
3 V(username)@F(auth.c:2)
4 V(result)@P(authenticate)@F(auth.c:15)
5 P(main)@F(auth.c:20)
```

Figure 1 Example of a *OntOSymbolTable* textual representation.

K represents a constant, used to represent local and global constants;
M represents a module, used to represent aggregations of functions or methods (objects or libraries for example).

A function called *tNormalize* was defined that is used to normalize program constructs' types:

$$tNormalize : term \longrightarrow PCType$$

that given a *term* (normally obtained from the *PseudoSymbolTable*), returns the *term PCType*. For example:

$$tNormalize(\text{"method"}) = P$$

$$tNormalize(\text{"function"}) = P$$

$$tNormalize(\text{"variable"}) = V$$

A *OntOSymbolTable* allows a more structured and systematic reasoning about program elements, also more information can be added to build more complex artifacts, more on this later. This definition is language agnostic, so far it was only used in the imperative programming paradigm [21], like C/C++ or Java, but minor tweaks can be required if other details characteristic to other paradigms need to be represented. This means that a front-end can be easily built for a specific language, or a compiler refactored, to create this table, and take advantage of all the features described in the next sections. Algorithm 1 summarizes how to build this table.

Algorithm 1 Create a *OntOSymbolTable* as a hash-map.

Require: $T : PseudoSymbolTable$
  $oost \leftarrow \{\}$ *// start with empty hash-map*
  for all $(id_t, type_t, scope_t) \in T$ **do**
    $type \leftarrow tNormalize(type_t)$
    $prt \leftarrow$ *{* parent PCid determined based on* $scope_t$ **}*
    $src \leftarrow$ *{* file and line number of identifier *}*
    $pss \leftarrow ProbSynSet(id_t)$
    $PCid \leftarrow id_t + +prt$
    $oost[\,PCid\,] \leftarrow (id_t, type, prt, src, pss)$
  end for
  return *oost*

A textual representation of the content of this table was also devised. Figure 1 is a small snippet of the *OntOSymbTab* calculated from a C source file `auth.c`. For example in Figure 1, line 1 is stated that a function name `authenticate` exists in file `auth.c` line `14`, or from Figure 1, line 4 that a local variable named `result` is defined in function `authenticate` in file `auth.c` line `15`.

The next section defines probabilistic synonymous sets, used in a $OntOSymbolTable$ and how they are calculated.

3.2 Probabilistic Synonym Set

In linguistics, and in the terminology discipline for a given term a list of synonyms can be built, this list is commonly called a *SynSet*, for more details about these sets please refer to [11] [12] [19]. For this work an extended version of this data structure called $ProbSynSet$ is defined. A list of triples composed by a term t that represents the same concept as the original term, a relation set r that represents the kind of relation between this term and the original one (synonym or translation for example), and a probability p which defines the degree of confidence that this term is actually conceptually equivalent to the original term.

$$ProbSynSet = Term \rightharpoonup Triple$$

$$\begin{array}{rcrcl} Triple & = & t & : & Term & \times \\ & & r & : & RelSet \in \{\{S\},\{T\}\} & \times \\ & & p & : & Confidence \in [0,1] & \end{array}$$

where,

t is a term (word) conceptually equivalent to the given term;

r is the relation that exists between t and the given term;

p is the degree of confidence that t is conceptually equivalent to the given term, $p \in [0,1]$.

The $ProbSynSet$ is calculated using a Probabilist Translation Dictionary (PTD), this can be seen as a common translation dictionary but with some important subtleties. First, PTDs are usually build from parallel corpora [19] which means that the language domain can be restricted to a specific domain. This is important because if translations are being used to find synonyms, in a software development context we wouldn't want for example *fork*, which can have many different meanings depending on context, representing a piece of cutlery, but a process (or similar concept). The second thing is the certainty level implied in PTD, that will be used to calculate the degree of confidence in the $ProbSynSet$.

A PTD can de defined as a finite function that given a $term$ returns a list of possible translations for $term$ and the degree of confidence that this translation is correct. More formally:

$$PTD = term \rightharpoonup PTDEntry$$

$$PTDEntry = t \rightharpoonup t \times p$$

where,

t is a possible translation of $term$;

p is the degree of confidence that t is a correct translation, $p \in [0,1]$;

This set ends up being a list of quasi-synonym, a list of terms that can represent the same concept. Besides, probability p gives a fine tuning capability to broad or short the scope of term gathering in this list. Typically a *cut line*, a way of saying that below this degree of confidence terms are to be ignored.

Figure 2 illustrates the $ProbSynSet$ calculated for term $username$. And Algorithm 2 describes how they can be calculated. In this algorithm a $WebService$ is being used to

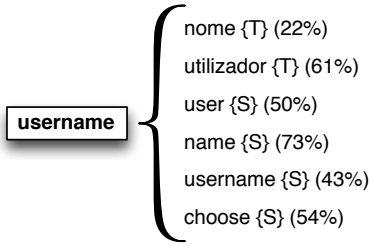

Figure 2 *ProbSynSet* for term *username*.

provide the required *PTDEntry*s. These resources were calculated and made available in the Per-Fide Project [2] environment.

Algorithm 2 Create a *ProbSynSet* for a given *term*.

Require: $term : String$
  $pss \leftarrow \emptyset$ // *start with empty hash*
  $ptd_{entry} \leftarrow WebService.PTD(term)$
  for all $k_i \in keys(ptd_{entry})$ **do**
    $(t_1, p_1) \leftarrow ptd_{entry}[\,k_i\,]$
    $pss[\,t_1\,] \leftarrow (t_1, \{T\}, p_1)$ // *add element to hash*
    $ptd^{-1}_{entry} \leftarrow WebService.PTD(t_1)$
    for all $k_j \in keys(ptd^{-1}_{entry})$ **do**
      $(t_2, p_2) \leftarrow ptd^{-1}_{entry}[\,k_j\,]$
      $pss[\,t_2\,] \leftarrow (t_2, \{S\}, min(p_1, p_2))$ // *add element*
    end for
  end for
  return pss

A more complete description of how PTDs are calculated and made available is out of scope for this article, please refer to references [18] [17] [16] for more details.

4 Relational Algebra

After calculating an *OntOSymbolTable* many information is available, including *ProbSynSet*s which are the base construct for creating semantic bridges between concepts. This section describes a set of algorithms that implement relational functions between *ProbSynSet*s. These functions are the minimal operations required to implement algorithms for relating terms and concepts. Figure 3 illustrates an example of comparing two terms: *username* and *user*, to determine if their *ProbSynSet*s are related, this could imply a semantic relation between these terms.

The first operation, that defines this relation, is the intersection ($\cap$) between *ProbSynSet*s. Although a *ProbSynSet* is a list, that can be seen as a set as defined by mathematics, intersection has to take in account probabilities and relation sets. This function is defined as:

$$\cap : ProbSynSet_A \times ProbSynSet_B \longrightarrow ProbSynSet_\cap$$

[2] http://per-fide.di.uminho.pt/

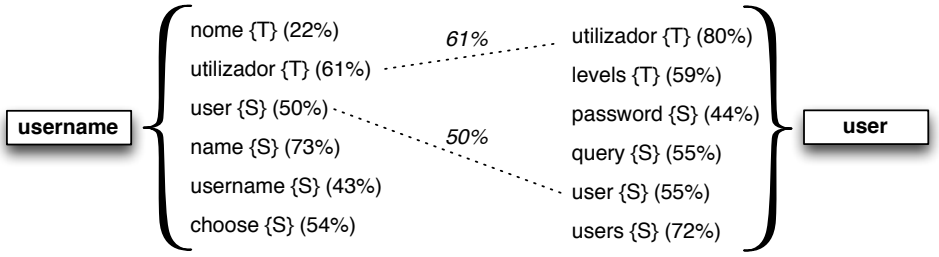

Figure 3 Relating *ProbSynSet*s for different terms.

Algorithm 3 describes how to calculate $ProbSynSetn_{\cap}$.

Algorithm 3 *ProbSynSet* intersection ($\cap$).

Require: $pss_A : ProbSynSet$
Require: $pss_B : ProbSynSet$
  $pss_{\cap} \leftarrow \emptyset$ *// start with empty hash*
  for all $k \in keys(pss_A)$ **do**
    $(t_A, r_A, p_A) \leftarrow pss_A[\,k\,]$
    if $t_A \in terms(PSS_B)$ **then**
      $(t_B, r_B, p_B) \leftarrow pss_B[\,t_A\,])$
      $r \leftarrow r_A \cup r_B$
      $p \leftarrow min(p_A, p_B)$
      $pss_{\cap}[\,t_A\,] \leftarrow (t_A, r, p)$ *// add element to hash*
    end if
  end for
  return $pss_{\cap}$

This operation can be used to claim that a non-empty intersection of *ProbSynSet*s implies that the terms are related.

$$ProbSynSet(t_A) \cap ProbSynSet(t_B) \neq \emptyset$$

$$\Rightarrow t_A \text{ is related to } t_B$$

This relations can mean that t_A and t_B are conceptually equivalent, if the degree of confidence in this relation is high enough. In order to be able to build ranks a similarity level function between two *ProbSynSet*s was defined (*simil*):

$$simil : ProbSynSet_A \times ProbSynSet_B \longrightarrow Float$$

Algorithm 4 shows how this function is implemented. This degree of similarity will be used in the next section to create ranks of conceptually equivalent suggestions.

Another operation required is the union of *ProbSynSet*s ($\cup$), defined as:

$$\cup : ProbSynSet_A \times ProbSynSet_B \longrightarrow ProbSynSet_{\cup}$$

This operation is used to join different *ProbSynSet*s in a single *ProbSynSet*. Algorithm 5 defines how this operation is implemented.

The next section shows how these operations can be used to implement algorithms for locating concepts in programs.

Algorithm 4 Similarity between two *ProbSynSets* (*simil*).

Require: $pss_A : ProbSynSet$
Require: $pss_B : ProbSynSet$
  $pss_\cap \leftarrow pss_A \cap pss_B$
  $similarity \leftarrow 0$
  for all $k \in keys(pss_\cap)$ **do**
    $(t, r, p) \leftarrow pss_\cap[\,k\,]$
    $similarity \leftarrow similatiry + p$
  end for
  return *similarity*

Algorithm 5 *ProbSynSet* union ($\cup$).

Require: $pss_A : ProbSynSet$
Require: $pss_B : ProbSynSet$
  $pss_\cup \leftarrow pss_A$ // *start with* pss_A *hash*
  for all $k \in keys(pss_B)$ **do**
    $(t_B, r_B, p_B) \leftarrow pss_B[\,k\,]$
    if $t_B \in terms(pss_U)$ **then**
      $(t_A, r_A, p_A) \leftarrow pss_A[\,t_B\,]$
      $p_B \leftarrow max(p_A, p_B)$
      $r_B \leftarrow r_A \cup r_B$
    end if
    $PSS_\cup[\,t_B\,], \leftarrow (t_B, r_B, p_B)$ // *set element in hash*
  end for
  return $pss_\cup$

5 Concept Location

Having defined basic operations to relate *ProbSynSet*s it is now possible to implement functions that make use of these basic operations to implement concept location oriented techniques. The most simple simple interesting function that can be implemented consists in a plain search of a concept in source code. Most of the times this is already possible, especially in modern development environments (Eclipse or Visual Studio for example). Commonly what these environments do is a previous collection of program constructs, and later when the programmer is trying to find a variable or function they do a simple pattern match between the term the programmer provides and the list of collected identifiers. This of course will not work if the programmer is using a term to look up a concept completely different from the one used by the original developer of the code. The advantage of this approach is that it tries to look up for conceptually equivalent terms, even if they were wrote using completely different words.

$$locate : OntOSymbolTable \times term \longrightarrow Rank$$

The *Rank* data structure is very simple, its' a list of pairs containing the program construct unique identifier *PCid*, and a positive real number representing the degree of similarity between the *ProbSynSet*s compared.

$$Rank = (PCid \times simil)*$$

Where,
PCid is the unique identifier for the program construct that matched as possible conceptually equivalent;
simil is the degree of similarity.
Algorithm 6 illustrates how the *locate* function can be defined.

Algorithm 6 Locate a *concept*.

Require: $oost : OntOSymbolTable$
Require: $term : String$
  $pss_C \leftarrow ProbSynSet(concept)$
  $rank \leftarrow \emptyset$ *// start with empty set*
  for all $pcid \in keys(oost)$ **do**
    $(id, type, prt, src, pss) \leftarrow oost[pcid]$
    $pss_\cap \leftarrow pss_C \cap pss$
    if $pss_\cap \neq \emptyset$ **then**
      $push(rank, (pcid, simil(pss_\cap))$ *// add element to set*
    end if
  end for
  return *rank*

This *locate* function is very simple, but more complex operations can be devised. For example, a *ProbSynSet* can be calculated by joining all the *ProbSynSet*s for all the program constructs found in a function, or code block. A real case scenario is for example having a function named f, that contains a variable named *username* and another named *password*. Although the identifier f by itself does not say much about what the function is dealing with, the variables *username* and *password* inside the function give a clue that is possible that this function is related with authentication or authorization. This means that joining the

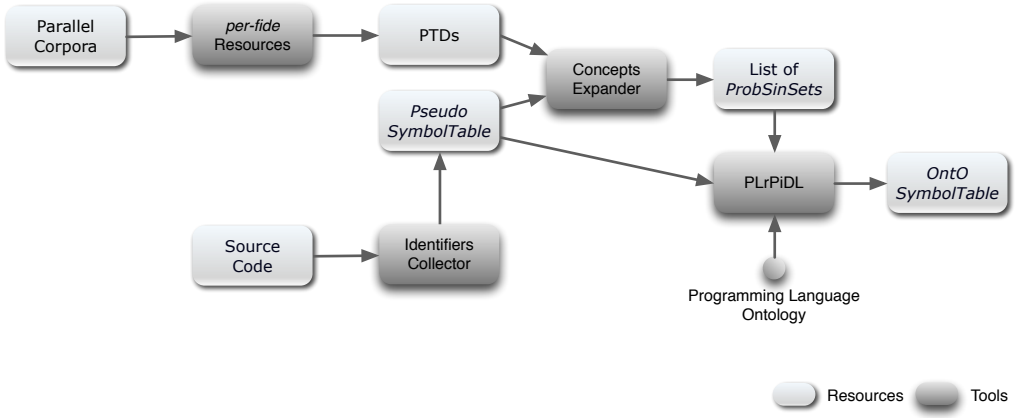

Figure 4 Web application architectural overview.

ProbSynSets for the identifiers found inside the function could probably suggest this, even that the name of the function by itself does not provide any significant semantic information. The required functions for *ProbSynSets* required to implement this algorithm were described in the previous sections: $\cup$ and *simil*. Similar reasoning functions could be implemented for code blocks, objects or even files.

6 Experimental Validation

In order to validate the approach described in this paper a set of tools were written to implement all the algorithms and functions described in the previous sections. This set of tools where then used to implement a web application prototype that using a browser would provide programmers an environment for locating concepts in programs.

The prototype application architectural overview is illustrated in Figure 4. The dark gray boxes represent tools that consume and create resources (which are represented in the light gray boxes), and the arrows represent the data flow of data between tools to produce the final *OntOSymbolTable*.

The *Identifiers Collector* is a tool that analyses source code and builds a *PseudoSymbol Table* that includes all the required information to create a *OntOSymbolTable*. This tool was implemented using *Exuberant Ctags* [3], that is normally used to create tag files that are used by editors or integrated development environments to implement features like auto-complete. This tool already provides support for a broad range of programming languages, this way it would be easier for our prototype to support all of those programming languages.

The *Concepts Expander* is a tool that can create *ProbSynSets*. It makes use of the *PTDs* created in the Per-Fide Project environment. These are built from parallel corpora automatically, and software related texts were used to try to constrain the vocabulary domain to the context of software development. Around *700 MB* of software related texts, for example documentation and user interface messages, in several languages were used to build the parallel corpora. This would help ensuring that the translation memories calculated terms were inside the software domain. The languages used were mainly portuguese and english, this means that the bag of conceptually equivalent terms in a *ProbSynSet* includes

[3] http://ctags.sourceforge.net/

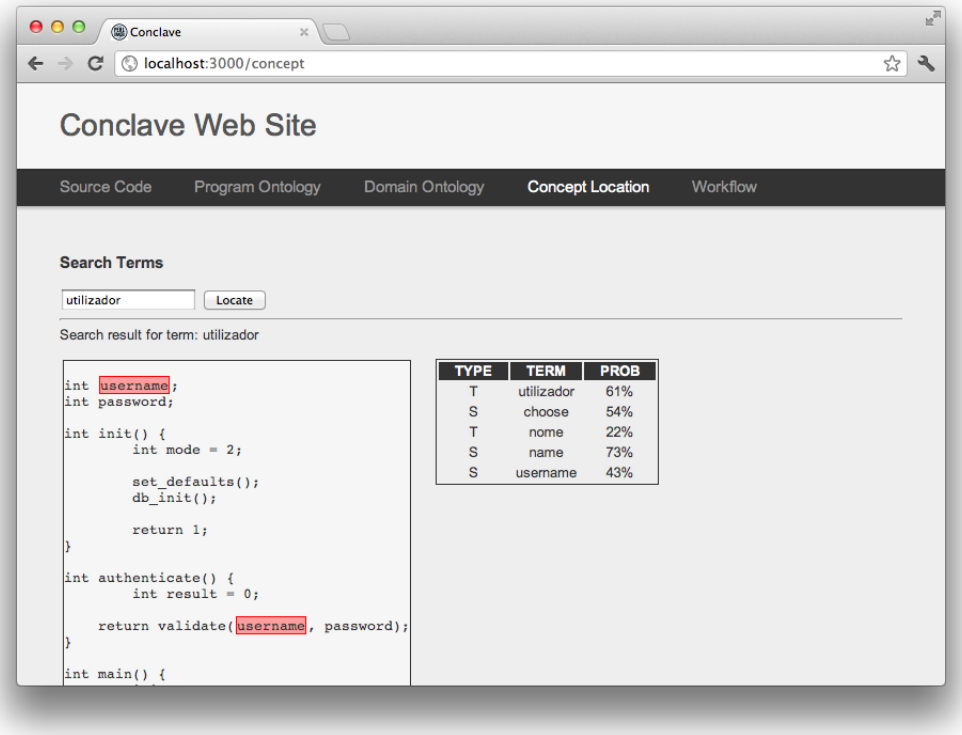

Figure 5 Web application screenshot.

portuguese and english terms. This is helpful if identifiers were not written in english, and the program maintainer is an english native speaker. A multilingual aware concept search is another advantage of this implementation. The more languages used to build the parallel corpora, the more languages can be supported.

Finally *PLrPidL* is a tool that creates a *OntOSymbolTable* for a given program, using the resources created by other tools. Figure 5 illustrates the application when searching source code for a term that was not in the original list of identifiers, but was found in a *ProbSynSet*, and therefor the application can highlight in the source code the terms that have high probability of representing the same concept the user was searching. The searched term *utilizador*, is a possible portuguese translation for *username*, which in addition to not being in the identifiers list is not even written in the same language.

These tools can also be used independently or composed in other workflows. Table 1 illustrates an initial study done with identifiers using these tools. The initial question that motivated this study was if the list of program identifiers is actually composed by words that a program maintainer would try to search for. The object of this study was TTH [4] a program that creates a HTML file from a LaTeX file.

Typically the first job of a programmer when acting as a maintainer for a given program is to actually find the source code responsible for implementing a part of the specification.

[4] http://hutchinson.belmont.ma.us/tth/

Table 1 Comparing Terms Found.

	# Identifiers Found	# *ProbSynSet* Created
TTH	925	925

	Strings	Words	%
Identifiers	476	55	12%
ProbSynSet	1659	1039	63%

Normally the maintainer needs to verify several areas of the code manually until the correct zone (or zones) that need update are found. The search usually starts with words that are related with the concept that needs to change, for example terms like *date*, *username* or *create* are common search keywords if for example trying to fix a bug in the creation date function. Most of the times this means that the programmer is using keywords in his natural language that in his domain are suitable candidates to represent the concepts that require changes. But, most of the times program identifiers are not so explicitly and many different names and versions of the same name can be used. For example looking at Table 1 the identifiers collector tool found 476 unique strings that identify program elements (variables, functions, etc). If this list of strings is checked in a typical dictionary, since the author of the code used english to write comments and the documentation the english dictionary of words was used, and we verified that only 55 of these strings were actually found. Well, if a programmer looking for source code zones to edit normally uses words in his domain language it will have a very low chance of success locating program elements that are related with the search key words. The *ProbSynSet* list of related terms was then used. A *ProbSynSet* is calculated for every identifier found in the program, this means that a total of 925 were calculated. Every *ProbSynSet* has a bag of terms that are conceptually equivalent to the original term found as identifier. This means that the scope of terms, linguistic speaking, was broaden. A total of 1659 strings are now available for the programmer to search for. And since 1039 of these strings are present in the dictionary the chances that the maintainer finds a word was greatly increased. This does not necessarily implies that the programmer will find the important zones of code to change, but will greatly improve the chances that a zone of code will be suggested based on key words search. In an information retrieval context we could say that the recall was greatly increased. An experiment that would verify if the precision is also increased is currently being devised.

7 Conclusion and Future Work

Several program comprehension techniques often rely on a mapping between program elements and more natural language terms that represent human oriented concepts. Program identifiers are a possible source of information about which specific source code areas are responsible for implementing these concepts. The meaningfulness of program identifiers directly influence the quality of the required mapping, and also future accuracy of concept locating tools.

The adoption of an ontology oriented table to represent identifiers, which describes not only the identifiers but also a set of more discovered information about them, provides an artifact that can be object of a systematic reasoning. Using this reasoning approach views of programs can be built that provide useful insight for locating code responsible for implementing concepts. Probabilistic *SynSet*s are an important element of this table that

provides anchors that can increase the recall concept location based on keywords search and similar approaches.

The implemented tools and current prototype help showing that concept location tasks recall and precision can be greatly increased by taking advantage of *OntOSymbolTable*. This ontology oriented constructs enriched table, can also be object of other processing functions using very simple and elegant basic operations to implement different analysis tasks.

Interesting tasks for further development of this work:

- combination of techniques described in the related work section, for example expand known abbreviations before calculating *ProbSynSet*s:
  - one example is using the work described in Section 2 to expand terms often written using abbreviations, or other shortening styles like camel case, this would allow the use of more accurate terms to build *ProbSynSet*s;
  - another example is adapting the algorithms described in this paper to allow the use of multi word terms, this would allow using identifiers that contain several words (sometimes joined with _ , for example *get_data*);
- experimental verification of the precision of recall of suggested areas of source code, similar studies like the one introduced in Section 6.

References

1 S.L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 156–159. IEEE, 2010.

2 A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 2007.

3 N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, page 4. IBM Press, 1998.

4 A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 205–214. IEEE, 2009.

5 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, 1993.

6 B. Caprile and P. Tonella. Restructuring program identifier names. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 97–107. IEEE, 2000.

7 F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

8 B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

9 E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 71–80. IEEE, 2009.

10 J.R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic extraction of a wordnet-like identifier network from software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 4–13. IEEE, 2010.

11 C. Fellbaum. Wordnet. *Theory and Applications of Ontology: Computer Applications*, pages 231–243, 2010.

12 J.J. Jiang and D.W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *Arxiv preprint cmp-lg/9709008*, 1997.

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

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

15 V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

16 A. Simoes, X.G. Guinovart, J.J. Almeida, and P. Natura. Distributed translation memories implementation using webservices. *Procesamiento del lenguaje natural*, 33:89–94, 2004.

17 Alberto Simões and José João Almeida. Parallel corpora based translation resources extraction. *Procesamiento del Lenguaje Natural*, (39):265–272, September 2007.

18 Alberto M. Simões and J. João Almeida. NATools – a statistical word aligner workbench. *Procesamiento del Lenguaje Natural*, 31:217–224, September 2003.

19 Alberto Manuel Brandão Simões. Parallel corpora word alignment and applications. Master's thesis, Escola de Engenharia - Universidade do Minho, 2004.

20 Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

21 D.A. Watt, W. Findlay, and J. Hughes. *Programming language concepts and paradigms*, volume 234. Prentice Hall, 1990.