The Journal of Systems and Software 100 (2015) 117–128

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

From source code identifiers to natural language terms

Nuno Ramos Carvalho[a,*], José João Almeida[a], Pedro Rangel Henriques[a], Maria João Varanda[b]

[a] Department of Informatics, University of Minho, Campus de Gualtar, 4710-057 Braga, Portugal
[b] Polytechnic Institute of Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal

ARTICLE INFO

Article history:
Received 2 October 2013
Received in revised form 29 July 2014
Accepted 9 October 2014
Available online 31 October 2014

Keywords:
Program comprehension
Natural language processing
Identifier splitting

ABSTRACT

Program comprehension techniques often explore program identifiers, to infer knowledge about programs. The relevance of source code identifiers as one relevant source of information about programs is already established in the literature, as well as their direct impact on future comprehension tasks.

Most programming languages enforce some constrains on identifiers strings (e.g., white spaces or commas are not allowed). Also, programmers often use word combinations and abbreviations, to devise strings that represent single, or multiple, domain concepts in order to increase programming linguistic efficiency (convey more semantics writing less). These strings do not always use explicit marks to distinguish the terms used (e.g., CamelCase or underscores), so techniques often referred as *hard splitting* are not enough.

This paper introduces Lingua::IdSplitter a dictionary based algorithm for splitting and expanding strings that compose multi-term identifiers. It explores the use of general programming and abbreviations dictionaries, but also a custom dictionary automatically generated from software natural language content, prone to include application domain terms and specific abbreviations. This approach was applied to two software packages, written in C, achieving a f-measure of around 90% for correctly splitting and expanding identifiers. A comparison with current state-of-the-art approaches is also presented.

1. Introduction

Understanding source code is a requirement for software maintenance and evolution tasks (Von Mayrhauser and Vans, 1995; Corbi, 1989). Software reverse engineering, is a process that aims to infer how a program works by analyzing and inspecting its building blocks and how they interact to achieve their intended purpose (Nelson, n.d.; Chikofsky and Cross, 1990). Many of these techniques rely on mappings between human oriented concepts and program elements (Rajlich and Wilde, 2002). Identifiers are one of the major source of information about program elements (Caprile and Tonella, 1999, 2000), and their meaningfulness has a direct impact on future comprehension tasks (Lawrie et al., 2006). Today, most of the programming communities promote the use of best practices and coding standards, that usually include rules and naming conventions which tend to improve the quality of identifiers used (e.g., the style guide for the Python programming language[1]).

Program identifiers have been greatly explored in the context of program understanding: for concept and concern location (see, e.g., Shepherd et al., 2007; Marcus et al., 2004; Abebe and Tonella, 2010; Liu et al., 2007), relating documentation with source code (see, e.g., Antoniol et al., 2002; Yadla et al., 2005; Marcus and Maletic, 2003), and other assorted software analysis applications (see, e.g., Lawrie et al., 2007; Lawrie and Binkley, 2011; Enslen et al., 2009; Carvalho et al., 2012, 2014). All this work can benefit from better program identifiers handling, and in many cases results can be improved (Dit et al., 2011).

Programming languages grammars constrain the strings that can be used as identifiers, not allowing spaces and other special characters (e.g., commas). These also tend to be short and easy to remember. Thus, acronyms and abbreviations are frequently used to represent real world concepts. The major goal of the work described in this paper, and related work (see Section 2), is to

* Corresponding author. Tel.: +351 253 604 430.
E-mail addresses: narcarvalho@di.uminho.pt (N.R. Carvalho), jj@di.uminho.pt (J.J. Almeida), prh@di.uminho.pt (P.R. Henriques), mjoao@ipb.pt (M.J. Varanda).

[1] Available from: http://legacy.python.org/dev/peps/pep-0008/ (Last accessed: 31-03-2014).

http://dx.doi.org/10.1016/j.jss.2014.10.013
0164-1212/

promote strings used as identifiers in the program domain, to sets of terms representing concepts in the application domain.

Identifiers created using a single word (or abbreviation) are easier to relate with domain terms. The real challenge are compound identifiers, i.e., identifiers assembled using more than one string (each representing a term), because these strings need to be correctly isolated before they can be linked with domain concepts. Moreover, these strings can be abbreviations or acronyms, and not actual words, increasing the tokenization process difficulty. Sometimes an explicit mark is used to delimit the strings used, for example, the identifier "insert_user" uses the underscore as an explicit mark to clearly distinguish the word "insert" and the word "user". Another common explicit technique is the CamelCase notation, for example in the identifier "insertUserData" the words used are explicitly delimited with an uppercase letter. This trend of explicit word compounds are referred in the literature as *hard splits* (or *hard words*). Many times no explicit mark is used to delimit the words, for example the identifier "timesort", was formed by joining the words "time" and "sort", but there is no explicit mark where one word ends, and the next word begins. This is usually referred as *soft splits* (or *soft words*). Splitting *soft words* is more complex that *hard words*, and the complexity increases when acronyms or abbreviations are used instead of complete words (Lawrie et al., 2006, 2006, 2007).

This paper introduces Lingua::IdSplitter (henceforth abbreviated LIdS),[2] a simple and fast algorithm that addresses the problem of splitting *soft words*, and can cope with 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. 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 major advantages of this approach is the use of a software specific dictionary computed automatically from the documentation corpus – built 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 terms specific to the application domain, not prone to be present in more general programming dictionaries.

To validate this approach, the technique was applied to sets of identifiers extracted from two open source projects written in C, using a heterogeneous combination of techniques for multi-word identifiers. The calculated sets of splits were compared with the manual split (traditionally called the *oracle*) and the overall splitting accuracy for several different settings was above 80%. To compare LIdS's performance against other state-of-the-art approaches, LIdS was also applied to other case studies available in the literature, in order to compare the achieved results.

The remainder of this paper is organized as follows: Section 2 discusses some related work; Section 3 describes the proposed approach to split and expand identifiers; Section 4 describes in detail the experimental study done to validate LIdS effectiveness; Section 5 relates and compares this work with state-of-the-art techniques that address the same problem; and finally, Section 6 presents some closing remarks and trends for future work.

[2] LIdS is available under GNU General Public License in the official comprehensive Perl network (CPAN) from: http://search.cpan.org/dist/Lingua-IdSplitter/ (Last accessed: 09-07-2014).

2. Related work

The work by Caprile and Tonella (1999), describes their lexical, syntactical and semantic analysis of function identifiers. In this work the creation of a dictionary based on information extracted from the software (source code mainly) was also a concern, and a valuable source of information. It also helps to highlight the relevance of NLP techniques applied in the context of Program Comprehension.

Enslen et al. (2009) describe Samurai, an automatic approach to split identifiers that uses a scoring function based on program-specific and global frequency tables. These tables are built by mining strings frequency in source code. The main intuition behind this algorithm is that sub-strings used as part of an identifier are likely to be used in other identifier from the same software, or even in other programs. A similar concern is behind our proposed custom corpus-based dictionaries, the expressions and terms found in natural language text belonging to the software domain are prone to be used as identifiers.

TIDIER (Madani et al., 2010; Guerrouj et al., 2011) is another approach for identifiers splitting. This algorithm is based in the Dynamic Time Warping algorithm, initially devised to compute distances in the context of speech recognition. And tries to achieve the correct split by computing distances between the identifier and words found in a set of dictionaries. This algorithm shares some concerns with LIdS, namely: (1) the use of dictionaries, including domain specific dictionaries, (2) the inference of abbreviations is based on computing some kind of metric between the identifier and words found in dictionaries. A possible short-coming of this approach (and the previous one – Samurai) is that both can produce a different split for the same identifier in different iterations. TIDIER also does not handle splitting identifiers that contain single letter abbreviations (e.g., *"gchord"*). LIdS given the same input, and the same set of dictionaries, always computes the same split/expansion.

TRIS (Guerrouj et al., 2012) is a more recent technique for splitting and expanding program identifiers proposed by the same authors of TIDIER. It also uses a set of dictionaries, general and domain specific. TRIS handles the splitting and expansion as an optimization problem, divided in two stages. During the first stage a set of dictionary word transformations is created including corresponding costs, and during the second phase the goal is to find the optimal path in the expansion graph. The resulting split and expansion corresponds to the one with the minimal cost.

The GenTest normalization algorithm proposed by Lawrie et al. (2010) and Lawrie and Binkley (2011) involves vocabulary normalization found in software artifacts (e.g., source code, documentation) to improve Information Retrieval software analysis tools. This algorithm starts by scoring all the possible splits, and the resulting split is the one with the highest score. The scoring function is based in a set of metrics, based on internal information (e.g., word characteristics), and external information (e.g., dictionaries).

LINSEN is an approach for splitting identifiers, and expanding abbreviations, proposed by Corazza et al. (2012). The authors propose the use of the Baeza-Yates and Perleberg, an approximate string matching technique, and the use of several general and domain specific dictionaries, to find a mapping between program identifiers and the corresponding set of dictionary words.

The work by Sureka (2012), is a more recent approach for splitting identifiers using the Yahoo web search and image search similarity distance. The main idea is that strings used as identifiers represent concepts in real life, and documents indexed in search engines include images and text, providing information to compute possible splits scores.

Butler et al. (2011) describe the INTT algorithm, a technique for identifiers names automatic tokenization, with special focus on

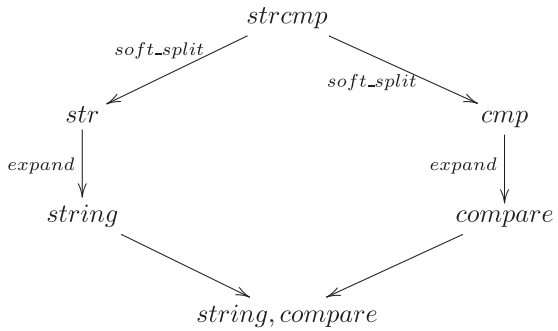

Fig. 1. Lattice for splitting the *strcmp* identifier.

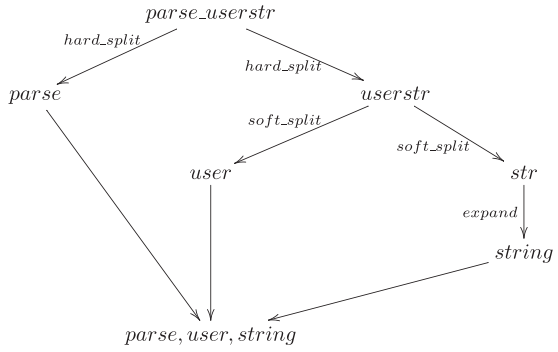

Fig. 2. Lattice for splitting the *parse _userstr* identifier.

single case identifiers, and identifiers containing digits. INTT also takes advantage of a pre-defined set of dictionaries, including commonly used abbreviations and acronyms.

These approaches (more details and other approaches in Feild et al., 2006; Dit et al., 2013) help to highlight the relevance of processing program identifiers, in the context of software analysis. The usage of NLP techniques and various types of dictionaries is a common trend in modern approaches, and the corresponding empirical studies help to highlight their added value and benefits.

3. The LIDS splitting approach

The goal of the technique described in this section is to split any combination of *hard* and *soft* terms (including abbreviations) found in an identifier. Fig. 1 illustrates the intended process. The identifier "strcmp" is composed of two abbreviations: "str" and "cmp", so this is the first level of intended split. The next improved step, is to start expanding abbreviations to the full term they represent. The best possible answer is to have the list of all correct terms: {*string*, *compare*}, in this example.

To cope with cases where combinations of *soft* and *hard* words are used, the algorithm first applies a *hard split* technique, followed by a *soft split* to the strings resulting from the first split. Fig. 2 illustrates an example.

3.1. The hard_split function

This function is responsible for splitting strings when an explicit separator mark is present. Since this is not the main focus of this work this is a simple function that only detects two explicit cases: special common characters[3] and the CamelCase notation. When these marks are found a simple split is made and the function returns the set of resulting strings. For splitting strings in CamelCase notation, the String::CamelCase (a Perl library) is used.[4]

Table 1
Dictionary valid words per string index for the identifier "timesort".

Index	Set	Index	Set
0	[*t*, *ti*, *time*, *times*]	4	[*s*, *so*, *sort*]
1	[*i*]	5	[*o*, *or*]
2	[*m*, *me*, *mes*]	6	[*r*, *rt*]
3	[*e*, *es*]	7	[*t*]

Algorithm 1 illustrates the *hard _split* function. The *matches* function (line 2 and 7) tries to match a string with a regular expression, returning a true value on success. *CamelCase* (line 7 and 8) represents a regular expression that matches the most common cases of CamelCase notation. The *split* function (line 3 and 8), returns a list of elements resulting from splitting a string using as delimiter a regular expression.

For example, the result of applying the *hard _split* function to the identifier "insert_UserDataStr" is the list: {*insert*, *user*, *data*, *str*}.

Algorithm 1. Compute *hard* splits.

```
Input: id : String                                   // identifier to split
Output: S : [String]
 1: L ← [id]
 2: if matches(id, special_marks) then
 3:    L ← split(id, special_marks)                  // special marks are: '.', '_' and '::'
 4: end if
 5: S ← ∅
 6: for all s_i ∈ L do
 7:    if matches(s_i, CamelCase) then
 8:       S ← S ∪ split(s_i, CamelCase)
 9:    else
10:       S ← S ∪ s_i
11:    end if
12: end for
13: return S
```

3.2. The soft_split function

After the *hard* words in the identifier are split, the next step is to split *soft* words. The *soft_split* function, given a string to split, returns a list of pairs, each pair containing the string representing the cut and the full term (in case of abbreviations were used for example). A simplified version of the algorithm is illustrated in Algorithm 2.

Lines 2–4 immediately return if the *id* to split is a valid string (i.e., a word or a known abbreviations[5]). Lines 6–14 compute all the possible valid strings that can be found starting in every position of the argument string. For example, Table 1 illustrates the possible strings per index for the "timesort" identifier. This means the set of valid words (according with the provided set of dictionaries) that start at every index. The actual computed set includes also the expanded terms (equal to the string if no abbreviation was used), and the weight assigned to the dictionary that validated the string. Once this set is computed, the next step (described in lines 16–20) is to build an automaton with all the words found, to calculate all the possible sequences of nodes (paths), that concatenate to rebuild the original identifier. An example of this automaton is illustrated in Fig. 3 for the "timesort" identifier.

The set of paths in the automaton define the set of string sequences that are candidates to be the identifier correct splits. The *post_process* function, called in line 21, allows for some extra candidates to be created. Currently, a new candidate is added to

[3] Currently these include: single dot, underscore and double colon.

[4] Available from: http://search.cpan.org/dist/String-CamelCase/ (Last accessed: 03-03-2014).

[5] Single letter strings (like "a" or "x") are valid English words. This technique can handle identifiers composed of such strings (e.g., "xyfigure" that splits to the set: {*x*, *y*, *figure*}).

Table 2
Top entries in the identifier "timesort" rank, sorted by score from highest to lowest.

Split	Score
{*time*, *sort*}	1.4400
{*ti*, *me*, *sort*}	0.1920
{*time*, *so*, *rt*}	0.1920
{*times*, *o*, *rt*}	0.1500
(...)	

the list when a sequence of 3 or 4 letters is found. The sequence of letters is added as a word with a weight lesser[6] than any dictionary, mainly to prevent over-splitting small unknown abbreviations and acronyms. Next, the algorithm computes the score for each candidate, creating a rank, where the top element (the sequence with the highest score) is returned as the resulting split. The top entries for the "timesort" identifier rank are illustrated in Table 2.

The *compute_word_graph* function, given a set of possible terms per original identifier index, builds an automaton (example illustrated in Fig. 3), and the *sort_by_score* function numerically sorts the candidates, using the entry scores, from highest to lowest. The *valid_term* and *score* functions are described in more detail later in this section.

Algorithm 2. Compute *soft* splits.

```
Input: id : String                                   // identifier to split
Output: S : [(String, String)]                       // list of pairs split,term
// return if valid word or know abbreviation
if (s, t, _) = valid_term(id) then
    return [(s, t)]
end if
// compute possible valid terms in id per index
terms ← ∅
for i = 0 to length(id) do
    for j = i to length(id) do
        str ← splice(i, j, id)
        if (s, t, w) = valid_term(str) then
            terms[i] ← terms[i] ∪ (s, t, w)
        end if
    end for
end for
// compute every possible sequence of terms
g ← compute_word_graph(terms)
candidates ← ∅
for all p_i ∈ paths(g) do
    candidates ← candidates ∪ p_i
end for
candidates ← candidates ∪ post_process(candidates)
// compute score for each candidate
scores ← ∅
for c_i ∈ candidates do
    scores{c_i} ← score(c_i)
end for
// sort candidates by score and select top ranked
rank ← sort_by_score(candidates, scores)
top ← pop(rank)
S ← map (fst) top
T ← map (snd) top
return zip(S, T)
```

3.3. The split function

Given an identifier this function computes a list of pairs (*string*, *term*) that represent the set of splits (and corresponding terms) for a single or multi-word identifier. It uses a combination of the *hard_split* and *soft_split* functions, and if a single word or known abbreviation is given as argument it returns the word or expanded abbreviation respectively.

[6] This weight is usually 0.1, since more specific dictionaries all have weights set to values above 0.1 (e.g., programming dictionary weight is 0.6, *custom corpus-based* dictionary weight is 0.6).

This function is the entry point for the technique and is described in Algorithm 3. It starts by applying an *hard_split* to the argument, and then applies a *soft_split* to every resulting string. The final result is a list of pairs: each containing the split, and the full expanded term.

Algorithm 3. Split an identifier.

```
Input: id : String                                   // identifier to split
Output: S : [(String, String)]                       // list of pairs split,term
S ← ∅; T ← ∅
hard_words ← hard_split(id)
for all s_i ∈ hard_words do
    (s, t) ← unzip(soft_split(s_i))
    S ← S ∪ s
    T ← T ∪ t
end for
return zip(S, T)
```

3.4. The valid_term function

The *valid_term* function, used by the *soft_split* function, decides for a given string and a set of dictionaries if the string is a valid term. A string is considered valid if present in any of the dictionaries. If valid, the function returns a tuple including: (1) the original string, (2) the term the string represents (if an abbreviation is used for example), and (3) the dictionary (that validated the string) weight. Algorithm 4 illustrates this function implementation.

Generally, a dictionary is defined as a pair: (1) a function that given a string returns a word, and (2) a weight:

$$Dictionary = words : String \rightarrow String \times weight : Float$$

The weight is a float that expresses the dictionary degree of confidence. This attributes main purpose is to give dictionaries a preference order. For example, the English language dictionary is always used, and has a weight set to less than more specific programming dictionaries. So that programming terms, more common to be used as program identifiers, have a higher chance to be included in the result of the identifier split. And also, terms that share the same abbreviation, can use expansions more specific to the program domain. For example, "directory" is commonly abbreviated as "dir", but in the *AbcMidi* package, "dir" is more often used to abbreviate "direction". The weight is also used to calculate the sequence score (more details on this in the next sub-section).

Algorithm 4. Verify if a string is a valid term.

```
Input: str : String                                  // term to be verified
Input: D : [Dictionary]                              // dictionaries set
Output: (s, t, w) : (String, String, Float)
for all d ∈ sort_by_weight(D) do
    if str ∈ domain(d{words}) then
        return (str, d{words}(term), d{weight})
    end if
end for
return ∅
```

3.5. The score function

The *score* function, is used by the *soft_split* function, to calculate a score for each possible sequence of strings (paths) found in the automaton. This score measures the likelihood that a given sequence of strings is the correct split for a multi-word identifier. The candidate sequences are sorted by score and the proposed solution is the sequence with the highest score.

The formula to calculate a score is analytically defined as:

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

where the multiplicand of factors (a factor is calculated for each element in the sequence) plus the length of the longer string in

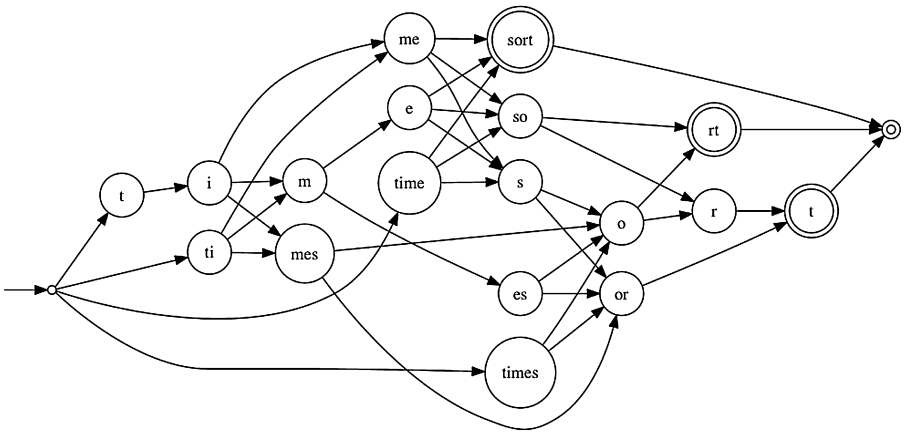

Fig. 3. Word automaton for the "timesort" identifier.

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.

Algorithm 5 illustrates the implementation of this function using these formulas. It simply iterates over the elements in the sequence, computing the multiplicand of element factors, adding the longest split length, and dividing by the squared sequence length.

Algorithm 5. The scoring function.

```
Input: S : [(String, String, Float)]                    // split,term,weight triple
Output: score : Float
prod ← 1
max ← 0
for all s_i ∈ S do
    prod ← prod × length(s_i{split}) × s_i{weight}
    if length(s_i{split}) > max then
        max ← length(s_i{split})
    end if
end for
score ← (prod + max) / length(S)^2
return score
```

3.6. Documentation corpus

Informally, a corpus is a collection of texts, usually representative of a given domain or subject. These constructs are used to build other common linguistic artifacts in the field of NLP (Halliday, 1992; Kennedy, n.d.; Martin and Jurafsky, 2000). The first step in order to build some of the dictionaries used by the *valid_term* function is to create the *documentation corpus*, by collecting (natural language) text from all the files included in the software package.

The corpus is created using a tool distributed with the DMOSS[7] framework. This tool iterates over the files in the package, and some specific file types are processed to extract their content as plain text to be included in the corpus. The following heuristics are currently being used:

- Documentation files (that can be plain text files or other common formats like HTML, *man* or JavaDoc) content is included in the corpus, specific format files are implicitly pre-processed for plain text extraction.
- Text from files commonly available in software packages that usually convey domain information is also included (e.g., *README*, *INSTALL* files).
- All other plain text files content is included. The file type computed by the *DMOSS-Oracle* tool – also distributed with the DMOSS framework – is used to decide which files are plain text.

All this content is stored in a plain text file called the *documentation corpus*, specific for each software package. This file is later processed to build more linguistic artifacts, namely dictionaries, as described in the next sub-section. Source code comments are extracted by a different process, and are not included in this corpus. More details about DMOSS are available in Carvalho et al. (2013) and Carvalho et al. (2014).

3.7. Custom corpus-based dictionary

Application domains tend to use a specific vocabulary, that uses terms, expressions and common abbreviations which are not easily found in general purpose dictionaries. This sub-section describes the technique devised to automatically create a dictionary for domain specific abbreviations and multi-word expressions from the documentation corpus, specific to each software package.

The starting points are: (1) the documentation corpus; (2) the set of program identifiers extracted from the source code; and (3) a general programming dictionary. The steps to create the *custom corpus-based* dictionary are:

1. Create the *srcIds* set, that includes the starting point (2) – the set of identifiers collected from the source code – and, the explicit identifiers found in the documentation corpus. In order to be considered an explicit identifier, the string needs to combine one or more terms using an explicit mark (i.e., use *hard words*).
2. Split the *srcIds* set using *hard* split techniques, the resulting set is called *SimpleIds – bag*.
3. Search possible identifier expansions in the corpus. For each string in the *SimpleIds – bag* calculate a set of regular expressions to extract probable expansions and multi-word correspondences (matches) in the corpus. Rank them by occurrence frequency.
4. For every multi-word expansions found, calculate the single word correspondences, and the non-trivial ones are added to the custom corpus-based dictionary. By non-trivial we mean exact matches (equal strings), and that are known words in English. For example, the multi-word identifier "timesig", that expands to

[7] DMOSS is a framework for software packages (mainly non-source content) analysis, available from http://search.cpan.org/dist/DMOSS/ (Last accessed: 27-03-2014).

Table 3
Derived regular expressions examples.

String	Derived regular expressions	Top Match	
mrest	`m\w*rest\w*`	*"multibar rest"*	√
	`m\w{,2}r\w{,2}est\w*`	∅	
timesig	`t\w*imesig\w*`	∅	
	`tim\w*esig\w*`	∅	
	`time\w*sig\w*`	*"time signature"*	√
chan	`c\w*han\w*`	*"chord handling"*	
	`c\w{,2}h\w{,2}an\w*`	*"channel"*	√

Table 4
Top entries in the identifier "timesort" rank, sorted by score from highest to lowest.

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

"time signature", produces: (1) "time" → "time" (trivial), and (2) "sig" → "signature" (non-trivial, hence added to the dictionary).
5. To create the final dictionary, expansions and multi-words are included based on a set of filters (concerned with increasing precision, even if lowering recall). Filters include, for example, minimum length for abbreviated strings (3 characters), rejecting abbreviated strings with vowels, and expansions with a length of 15 characters or more.

For example, in the context of the *AbcMidi* package, many compound identifiers are found (e.g., "mrest" and "timesig"), and abbreviated terms (e.g., "chan"). Table 3 illustrates some example regular expressions created automatically to search the package corpus for the corresponding expansions. The top match for each expression is also illustrated, and the final expansion selected, either by filters or frequency count (when different expansions are available). The heuristics to create the regular expressions involve iteratively filling gaps between characters with wildcards and spaces.

The first ranked corespondent occurrences in the corpus are: "multibar rest", "time signature" and "channel" respectively. The final corpus based dictionary (after all the inference process) includes: expansions (e.g., "chan" → "channel"), abbreviations (e.g., "flg" → "flag"), multi-word (e.g., "timesig" →{*time*, *signature*}), some words not present in a general English dictionary but valid in the application context (e.g., "lynx") (Table 4);

The automatic creation of these dictionaries, which provide valuable information for splitting source code identifiers, is one of the major novelties introduced by this work. These allow the correct split and expansion of strings difficult to achieve otherwise (e.g., the "*hornp*" expands to "hornpipe". And increases the relevance of specific domain terms, not found in general programming dictionaries (e.g., "*anacrusis*" or "*accidentals*", from the *AbcMidi* corpus) but frequent in this package as identifiers. This dictionary is used by the *valid_term* function, which allows the *soft_split* function to handle (single and multi-word) domain abbreviations found in identifiers.

3.8. Other dictionaries

Since the custom corpus-based dictionary goal is to capture application domain specific vocabulary, more general programming common abbreviations and acronyms may not be present in this dictionary. Also, documentation may not be available to create the documentation corpus. To overcome this and similar situations another set of dictionaries are being used by LIDS:

programming: includes some general programming terms and abbreviations (e.g., "msg" → "message", "param" → "parameter") that have been collected over time by the authors, it has around 110 entries.
acronyms: a set of well known and common acronyms (e.g., HTML, XML, BSD, SQL), this dictionary has around 130 entries. These acronyms are not expanded to the full set of terms in this dictionary. In order to keep the setting for the experimental verification described in Section 5 as close as to the one described in Guerrouj et al. (2012), the acronyms dictionary provided by Guerrouj et al. was used to create the first version of this dictionary.
abbreviations: includes common abbreviations general to most programs (e.g., "ctrl" → "control", "buff" → "buffer"). Again, to keep the setting as close as possible to the one described in Guerrouj et al. (2012), the abbreviations dictionary provided by Guerrouj et al. is used. It has around 190 entries.
general: dictionary for the English language, the dictionary provided by the aspell tool[8] is used. It has around 120,000 entries.

4. Experimental validation

In order to measure its ability to correctly split and expand identifiers, LIDS was applied to two open source (so that the code is readily available) software packages: *tree* (version 1.5.3),[9] that implements the *tree* command, which can be used to list directories content hierarchically; and *AbcMidi* (version 2012.12.25),[10] a package that provides a set of tools to convert ABC[11] files to the MIDI format. This last package was also chosen as case study, because it acts on a specific domain (music), that has a specialized vocabulary, which terms are prone to be used as program identifiers.

Both packages are written in C. Source code written in this particular language was chosen for the experimental validation because although this language is being used for many years, there in no mainstream tradition for the techniques used to compose multi-word identifiers. Typically, many combinations of techniques are used. This is not the case for other languages, like Java for example, where there is a more traditional habit to use CamelCase for example Guerrouj et al. (2011). Another relevant detail about these packages is they are quite old, and different programmers have changed the code, increasing the heterogeneity of ways to create identifiers (either by composition or abbreviation). These characteristics make the splitting process (even manually) harder, but allow to better conclude about the ability of the proposed technique to generalize for other software packages.

The goal of this experiment is to measure LIDS ability to correctly split and expand program identifiers. The following research questions (RQ) 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?
- **RQ3:** What is the gain of using the custom corpus-based dictionary when splitting and expanding identifiers with LIDS?

To answer these questions the splits and expansions computed by LIDS were compared with the correct split obtained from the *oracle* (the *correct* answer). Besides splitting correctness, the set

[8] Available from: http://aspell.net (Last accessed: 12-02-2014).
[9] Available from: http://mama.indstate.edu/users/ice/tree/ (Last accessed: 01-10-2013).
[10] Available from: http://abc.sourceforge.net/abcMIDI/(Last accessed: 01-10-2013).
[11] A text notation to represent music.

Table 5
Some software packages characteristics.

Package	Files	KLOC[a]	Identifiers		Oracle	
			Total	Multi-word	Splits	Terms
tree	10	~2	235	145 (62%)	161	147
AbcMidi	86	~33	3437	2142 (62%)	1644	1565

[a] Thousands Lines of Code.

of calculated full terms was also compared, to validate the ability of expanding strings to terms in case abbreviations were used. Accuracy, precision and recall measurements were made in three different settings:

- *HardSplit*, in this setting only LIDS *hard_split* function is called, this acts as the baseline for other comparisons. No dictionaries are used by this function.
- *Split*, in this setting LIDS *split* function is used to compute splits and term sets. In this setting the following dictionaries are used: *programming*, *acronyms*, *abbreviations*, and *general* (details about these are discussed in the previous section).
- *CorpDict*, is equivalent to the previous setting, but the custom corpus-based dictionary, built for each specific package, is also included in the dictionary set.

The identifiers correct split and abbreviations expansion are required for the evaluation, the next section describes the creation of the *oracle*.

4.1. Creating the oracles

The *oracle* consists of two sets for each analyzed package: (1) the correct split, the list of strings in which a multi-word identifier is correctly split; and (2) the correct terms set, the list of terms that compose a multi-word identifier (in this set abbreviations are expanded). Single terms are also included in the set, because although the split is straightforward, the string used can still be an abbreviation. The steps to build the *oracle* were:

1. Collect identifiers from source code files.
2. Remove identifiers with two or less characters.
3. Remove duplicate identifiers.
4. For each identifier in the set, by analyzing the source code, we: (a) manually created the correct split for the identifier (e.g., the correct split set for the identifier "wcount" is: {*w*, *count*}); and, (b) manually created the correct set of intended terms by the original programmer (e.g., the set of terms for "wcount" is: {*word*, *count*} – this example function counts number of words).

Table 5 includes some characteristics about the software packages (number of files, and number of lines of code); and about identifiers found in each package: total number of identifiers, percentage of identifiers that are composed of several terms, and the number of unique identifiers in each *oracle*. In some cases there was not a general consensus between the authors on how to split an identifier or expand an abbreviation, these cases were not included in the *oracle*. Mainly to try to reduce the final number of errors present in the *oracle*.

4.2. Accuracy

Accuracy measures the ability to correctly split and expand the terms that compose an identifier. The function that validates if the split is correct returns a binary value: 1 in case the set of splits (or terms) is exactly equal to the set in the *oracle*, and 0 otherwise. It correctly measures the algorithm overall accuracy, but with a small draw-back: in case the algorithm misses one correct single string (or expansion) in the set, but all the others are correct, the validation function returns 0, even if some partial result is correct.

Table 6
Correct split accuracy means and correct terms accuracy means for *tree* and *AbcMidi*, in the three settings.

Setting	Measure	*tree*		*AbcMidi*	
		Splits	Terms	Splits	Terms
HardSplit	Accuracy	0.4907	0.2721	0.3668	0.3073
Split	Accuracy	0.8571	0.6939	0.8832	0.7885
CorpDict	Accuracy	0.8696	0.7007	0.9300	0.8281

Table 7
Precision, recall, and f-measure means, for correct splits and correct terms sets, for *tree* and *AbcMidi* packages, in the three settings.

Setting	Measure	*tree*		*AbcMidi*	
		Splits	Terms	Splits	Terms
HardSplit	Precision	0.5031	0.4150	0.4218	0.3887
	Recall	0.5021	0.4138	0.4034	0.3754
	f-Measure	0.5025	0.4143	0.4107	0.3807
Split	Precision	0.8834	0.7973	0.9230	0.8782
	Recall	0.8903	0.8033	0.9307	0.8856
	f-Measure	0.8858	0.7992	0.9257	0.8810
CorpDict	Precision	0.8959	0.8041	0.9548	0.9100
	Recall	0.9027	0.8101	0.9552	0.9112
	f-Measure	0.8982	0.8060	0.9544	0.9101

Table 6 illustrates the results of validating the accuracy measure on both packages, in the different settings for each set of unique identifiers in the corresponding oracle. Using the *CorpDict* setting, LIDS achieved an accuracy mean of 0.8696 when splitting identifiers, i.e., around 86% of the identifiers were split correctly; and an accuracy mean of 0.7007 when splitting and expanding identifiers, i.e., around 70% of the identifiers were correctly split and expanded to the set of terms in the *oracle* for the *tree* package. The same setting achieved a mean of 0.9300 for splitting terms, and 0.8281 for splitting and expanding terms for the *AbcMidi* package.

4.3. Precision and recall

To overcome the draw-back of measuring using the binary validation function described in the previous sub-section, a precision and recall measure of the correct splits (and terms) was also made.

For a given identifier *id* to split let the *oracle* split set be: $o = \{o_1, o_2, \ldots, o_n\}$, and $s = \{s_1, s_2, \ldots, s_n\}$ 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 formulas are applied when calculating the measures for correct terms, but using the calculated sets of terms instead of splits.

Once precision and recall are computed the f-measure can also be calculated. This measure represents a weighted average between precision and recall, and is calculated using the following formula:

$$\text{f-Measure} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Table 7 summarizes the precision, recall and f-measure means for correct splits and sets of terms in the different settings for both packages. Using the *CorpDict* setting, LIDS achieved a precision mean of 0.8959 and a recall mean of 0.9027 when splitting identifiers for the *tree* package; and a precision mean of 0.9548 and a recall mean of 0.9552 when splitting identifiers for the *AbcMidi*

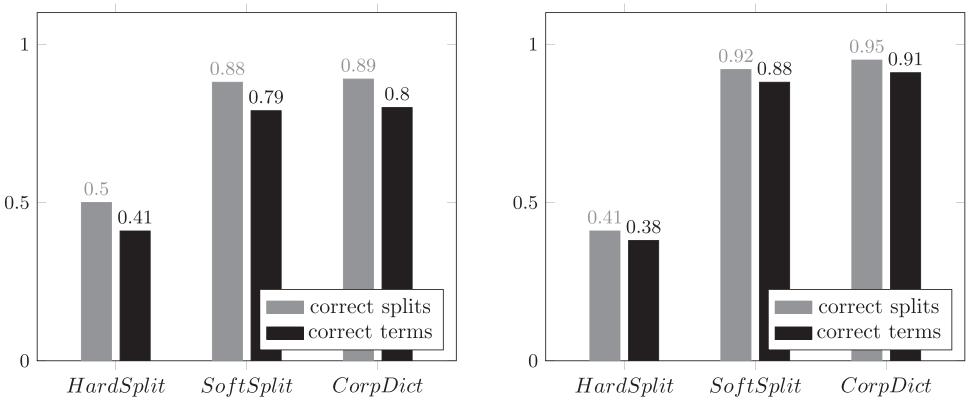

Fig. 4. f-Measure means for correct splits and correct terms sets, for the *tree* (left) and for the *AbcMidi* (right) packages.

package. For the correct set of terms, in the same setting, LIDS achieved a precision mean of 0.8041 and a recall mean of 0.8101 for the *tree* package, and a precision mean of 0.9100 and a recall mean of 0.9112 for the *AbcMidi* package. A f-measure mean of 0.8060 was achieved when splitting and expanding terms of the *tree* package, and a f-measure mean of 0.9101 for the *AbcMidi* package.

Fig. 4 illustrates f-measure means for correct splits, and correct terms sets for the *tree* and for the *AbcMidi* package.

4.4. Results discussion

Regarding **RQ1** and **RQ2**, the results obtained indicate that the proposed technique performed well in the analyzed programs, written in C, and that use an heterogeneous combination of techniques to create program identifiers (f-measure means in Table 7 illustrate this). The *HardSplit* setting achieves, at best, a f-measure mean of 0.5025 when splitting identifiers for the *tree* package. This setting only splits *hard words*, which clearly is not enough for the analyzed packages identifiers sets. The *Split* setting helps to illustrate that LIDS outperforms a simple hard splitting technique, accuracy means in Table 6 and f-measures means in Table 7 support this statement. Taking advantage of the custom corpus-based dictionary (the *CorpDict* setting) improves all the results, mainly because it introduces package specific abbreviations, i.e., abbreviations not found in the *abbreviations* or *programming* dictionaries (e.g, "ana" → "anacrusis", "syll" → "syllable"). This helps to answer **RQ3**, the empirical data shows that for every setting, accuracy, precision and recall means increase when using the custom corpus-based dictionary in the analyzed packages. This increase is higher for the *AbcMidi* package, mainly because the corpus is bigger (∼30,000 words, versus the ∼2000 words for the *tree* package), and it includes a more specialized vocabulary, allowing for the custom corpus-based dictionary to capture a set of terms more representative of the application domain.

The main reasons for not reaching 100% precision, in the *HardSplit* setting is straightforward: the analyzed software packages has many multi-term identifiers composed of *soft words*. Regarding the LIDS approach, the main reasons for failing splits are: over-splitting, splitting abbreviations or expressions used by the developers that are not present in the dictionaries (e.g., splitting "downoct", in the set {*down*, *o*, *ct*}, and the set in the *oracle* is {*down*, *oct*}); unexpected words found in the identifier that are validated by the general English dictionary (e.g., the *oracle split set for* "gotends" is {*got*, *ends*}, but LIDS resulting split set is {*go*, *tends*}); and invented words by the programmer that are not valid in any dictionary but are reasonably perceived by humans (e.g., "chording" is used to represent the action of making chords, but this is not a valid English word, so its split set ends up being erroneously {*c*, *hording*}). The LIDS algorithm can cope with an heterogeneous set of dictionaries, and the most natural solution to address these issues is to devise methods for creating new and improved dictionaries, that can better capture the specifics of the vocabulary used by programmers in software development. Which means that improving the results is possible without changing the algorithm itself, but providing more accurate dictionaries. Furthermore, these improved dictionaries can be used by other dictionary based approaches for identifiers splitting.

Although all the measures illustrated in the previous sub-section show that LIDS performed well for the analyzed programs, there is still not enough evidence to generalize its effectiveness for all programs, or other programming languages. The data analyzed and the results presented along this section are available online from http://conclave.di.uminho.pt/articles/, including all the dictionaries and oracles used.

4.5. Threats to validity

One shortcoming of the validation approach is the existence of errors in the *oracle*, either errors by typos or misspelling, or because the manual approach (split or term expansion) was not exactly the same as the intent of the original programmer. When manually creating the oracle some terms were not included because there was not a clear consensus amongst the authors on how to split or expand a given identifier. Although these cases would provide good examples to evaluate the performance of the splitting technique, we were afraid to end up including errors in the oracle, which would end up by jeopardizing the evaluation results.

Another shortcoming of the evaluation is that sometimes the exact split or term chosen by the algorithm is not syntactically equal to the manual split but semantically equivalent, this is the case of plurals (e.g., "chord" versus "chords"), or transitive verbs (e.g., "trim" versus "trimming"). The evaluation uses a syntactic exact match, meaning that all these examples result in a incorrect split/expansion. This issue is mainly related with the evaluation of the resulting splits and terms.

Another shortcoming of this experiment is the fact that it was applied to a couple of software systems only, with a reduced number of identifiers. Therefore, the set of analyzed identifiers is not

Table 8
JHotDraw and *Lynx* main characteristics.

	JHotDraw	*Lynx*
Release	5.1	2.8.5
Files	155	247
Size (KLOC)	16	174
Identifiers (≥2 chars)	2348	12,194

Table 9
Precision, recall, and f-measure for several approaches on JHotDraw.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	Camel Case	1.0000	1.0000	0.9244	1.0000	0.2424
	Samurai	1.0000	1.0000	0.9316	1.0000	0.2244
	TIDIER	1.0000	1.0000	0.9716	1.0000	0.1472
	TRIS	**1.0000**	**1.0000**	**0.9804**	**1.0000**	**0.2025**
	INTT	1.0000	1.0000	0.9623	1.0000	0.1704
	LIDS	1.0000	1.0000	0.9591	1.0000	0.1728
Recall	Camel Case	1.0000	1.0000	0.9203	1.0000	0.2502
	Samurai	1.0000	1.0000	0.9367	1.0000	0.2129
	TIDIER	1.0000	1.0000	0.8984	1.0000	0.2158
	TRIS	1.0000	1.0000	0.9084	1.0000	0.1213
	INTT	1.0000	1.0000	0.9606	1.0000	0.1760
	LIDS	**1.0000**	**1.0000**	**0.9641**	**1.0000**	**0.1583**
f-Measure	Camel Case	1.0000	1.0000	0.9217	1.0000	0.2476
	Samurai	1.0000	1.0000	0.9325	1.0000	0.2200
	TIDIER	1.0000	1.0000	0.9233	1.0000	0.1791
	TRIS	1.0000	1.0000	0.9328	1.0000	0.1614
	INTT	**1.0000**	**1.0000**	**0.9607**	**1.0000**	**0.1733**
	LIDS	1.0000	1.0000	0.9603	1.0000	0.1670

enough to generalize the results obtained for these specific software systems to all software packages.

5. Experimental comparison

In order to verify the performance of LIDS against other state-of-the-art approaches two more experiments were conducted. The following research question was defined:

- **RQ4:** What is the performance of LIDS compared with other state-of-the-art approaches for splitting and expanding identifiers?

To compare the performance of several techniques, all the approaches need to be applied in the same setting: same program identifiers, same *oracle*, same measures, etc. To hold true to these assumptions, instead of devising new experiences, some case studies described in Guerrouj et al. (2012) and Hill et al. (2013) were used. The data provided by the authors includes the *oracles*, allowing other approaches to compute splits on the same identifiers, and assume the same correct answers. The goal of the following experiences is to re-create the original experience, but including the LIDS approach. All the LIDS results achieved in this section use the *CorpDict* setting, described in Section 4; unless stated otherwise, the documentation corpus and custom corpus-based dictionary were created for each analyzed package.

5.1. First experiment

The subjects for the first experiment, described in Guerrouj et al. (2012), are two programs: *JHotDraw*,[12] a framework for technical and structured graphics, written in Java; and *Lynx*,[13] a text-based web browser written in C. Both projects are open-source, so the source code is readily available. Table 8 highlights some characteristics about these packages.

The study follows the design described by Guerrouj et al. (2012), so that the results can be compared. The main independent variable is the approach used to compute the split and expansion set, which is compared to the gold set provided by the *oracle*. LIDS was applied to the identifiers from the *oracle* for each package, precision, recall and f-measure were computed for the resulting splits. Tables 9 and 10 summarize the results found in Guerrouj et al. (2012), including two new approaches: LIDS and INTT. This data helps to answer **RQ4**, comparing the ability to split and expand identifiers, the new approaches f-measures means values are close to the ones from TRIS and TIDIER. For the *JHotDraw* package, LIDS achieved a f-measure mean of 0.9603, which is very close to TRIS and TIDIER. For the *Lynx* package, LIDS achieved a f-measure mean of 0.8593, lower than TRIS f-measure 0.9206, but close to TIDIER f-measure mean 0.8525. Results are better for the *JHotDraw* package mainly because of the same reasons already highlighted in Guerrouj et al. (2012): this package follows coding standards and most of the identifiers are composed of hard words, opposed to the *Lynx* project, where a more ad-hoc set of rules were used to create multi-term identifiers, and hard words were less used.

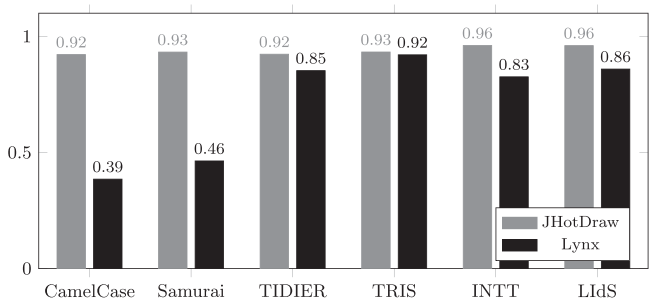

Fig. 5. f-Measure means for *JHotDraw* and *Lynx* using different approaches.

The plot in Fig. 5 illustrates the f-measure means for both packages, using the different approaches. For the *JHotDraw* package all the values are above 0.90, but for *Lynx*, there is a clear gap between CamelCase and Samurai approaches, and TIDIER, TRIS, LIDS and INTT approaches. Mainly because *Lynx* identifiers use more techniques to shorten identifiers, and create abbreviations, harder to split by techniques best suited to splitting hard words.

5.2. Second experiment

The next experiment, based on Hill et al. (2013), aims to compare the techniques ability to split a multi-term identifiers strings. In their paper Hill et al. use the LUDISO oracle which contains a set of 2731 identifiers from a collection of 2117 open source program written in C, C++ and Java; and the manual splits created by human annotators. State-of-the-art approaches are applied to the identifiers in the oracle, and their ability to correctly split the identifiers set is measured by means of accuracy, precision and recall,

[12] Available from: http://www.jhotdraw.org/ (Last accessed: 07-03-2014).
[13] Available from: http://lynx.isc.org/ (Last accessed 07-03-2014).

Table 10
Precision, recall, and f-measure for several approaches on Lynx.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	Camel Case	0.0000	0.5000	0.4065	0.7500	0.4147
	Samurai	0.0000	0.5000	0.4767	1.0000	0.4089
	TIDIER	0.8000	1.0000	0.8609	1.0000	0.2674
	TRIS	**1.0000**	**1.0000**	**0.9344**	**1.0000**	**0.1369**
	INTT	0.7500	1.0000	0.8294	1.0000	0.3215
	LIDS	0.8000	1.0000	0.8539	1.0000	0.2868
Recall	Camel Case	0.0000	0.3333	0.3705	0.6667	0.4066
	Samurai	0.0000	0.3333	0.4569	1.0000	0.4101
	TIDIER	0.7500	1.0000	0.8499	1.0000	0.2684
	TRIS	**1.0000**	**1.0000**	**0.9138**	**1.0000**	**0.2060**
	INTT	0.7500	1.0000	0.8244	1.0000	0.3269
	LIDS	1.0000	1.0000	0.8681	1.0000	0.2711
f-Measure	Camel Case	0.0000	0.4000	0.3851	0.7273	0.4086
	Samurai	0.0000	0.4000	0.4634	1.0000	0.4084
	TIDIER	0.6667	1.0000	0.8525	1.0000	0.2664
	TRIS	**1.0000**	**1.0000**	**0.9206**	**1.0000**	**0.2055**
	INTT	0.7500	1.0000	0.8258	1.0000	0.3245
	LIDS	0.8333	1.0000	0.8593	1.0000	0.2796

and analyzed by groups (e.g., programming language, identifiers subsets).

Accuracy is a binary measure (as before), if the technique output split is exactly equal to the corresponding split in the oracle the accuracy value is 1, but if there is any difference between the technique output and the oracle the accuracy value is 0. For this experiment the *programming*, *acronyms*, and *abbreviations* dictionaries described in previous sections were used. The oracle is composed from thousands of programs, it was not feasible to compute the custom corpus-based dictionary for every software package. Instead, a documentation corpus was created that includes the natural language text for the top three programs found in the oracle (*mozilla-source*, *mysql* and *cinelerra*), a custom corpus-based dictionary was created from this corpus and used by LIDS.

Table 11 mirrors the results compiled in Hill et al. (2013) but includes the LIDS approach results, which achieved an accuracy mean of 0.67 when splitting the identifiers from the LUDISO oracle. The major reason for LIDS accuracy to be lower than most of the other approaches is over-splitting. For example, the result of splitting "GGGPP_CDMA2000" with LIDS is the set {*ggg*, *pp*, *c*, *dma*, 2000}, the oracle correct split is {*GGGPP*, *CDMA*, 2000} the excessive split occurs because the strings "GGGPP" and "CDMA" and not present in any dictionary used. The only way to overcome similar situations is using the custom corpus-based dictionary to gather such strings, but this dictionary was not created for the package where this identifier was extracted from. Although, there is no assurance that every abbreviation string would be added to every package custom corpus-based dictionary, at least some of them would be expected to, increasing the approach overall accuracy.

Table 11
Mean per-identifier accuracy for each programming language subset for the LUDISO oracle.

Technique	All	*C*	*C++*	*Java*
Samurai_all	0.82	0.79	0.85	0.83
Samurai_cpp	0.81	0.77	0.85	0.81
GenTest_lg_all	0.80	0.78	0.82	0.78
INTT	0.75	0.70	0.78	0.78
GenTest_med_java	0.74	0.75	0.77	0.71
CS	0.71	0.68	0.72	0.72
DTW	0.69	0.75	0.66	0.65
LIDS	0.67	0.69	0.72	0.60
Greedy_lg	0.60	0.59	0.66	0.54
Greedy_sm	0.56	0.58	0.59	0.51
Count	2663	885	887	891
% of data	100%	33%	33%	33%

Besides accuracy, precision and recall measures were also made. These are calculated in a slightly different way than in the previous experience, since the goal now is to measure the correct splits, and not the resulting set of terms. Table 12 summarizes the intra-technique results for precision and recall using different approaches presented in Hill et al. (2013), but including the LIDS approach. The newly introduced technique results achieved a precision mean of 0.90, a recall mean of 0.96, which translated in a f-measure mean of 0.92. These results are close and in line with other approaches. The main reason for having a precision under

Table 12
Mean precision (P), recall (R), and f-measure (F) for each programming language subset for the LUDISO oracle, sorted by mean overall accuracy.

Technique	Measure	All	*C*	*C++*	*Java*
Samurai_all	P	0.97	0.96	0.98	0.98
	R	0.96	0.94	0.97	0.97
	F	0.96	0.95	0.97	0.97
Samurai_cpp	P	0.98	0.98	0.98	0.98
	R	0.95	0.93	0.96	0.96
	F	0.96	0.94	0.97	0.97
GenTest_lg_all	P	0.97	0.97	0.97	0.96
	R	0.96	0.94	0.96	0.98
	F	0.96	0.95	0.96	0.96
INTT	P	0.98	0.99	0.99	0.98
	R	0.93	0.91	0.94	0.95
	F	0.96	0.95	0.95	0.96
GenTest_med_Java	P	0.95	0.97	0.98	0.98
	R	0.97	0.94	0.97	0.95
	F	0.95	0.95	0.97	0.96
CS	P	1.00	1.00	1.00	1.00
	R	0.90	0.88	0.90	0.91
	F	0.94	0.92	0.94	0.95
DTW	P	0.93	0.94	0.92	0.92
	R	0.94	0.96	0.91	0.95
	F	0.93	0.95	0.91	0.93
LIDS	P	0.90	0.90	0.91	0.87
	R	0.96	0.96	0.97	0.96
	F	0.92	0.92	0.93	0.91
Greedy_lg	P	0.89	0.89	0.91	0.86
	R	0.97	0.96	0.98	0.98
	F	0.92	0.91	0.94	0.91
Greedy_sm	P	0.88	0.89	0.90	0.86
	R	0.97	0.96	0.98	0.98
	F	0.92	0.91	0.93	0.91

100% is the over-splitting introduced by splitting specific abbreviations not presented in any dictionary as discussed previously. This data helps to answer **RQ4**, LIDS ability for correctly split identifiers is in line with other approaches.

5.3. Threats to validity

Regarding both comparison experiments, the major threats to validity are concerned with the oracles, and how the resulting splits are compared. Even if a lot of effort is dedicated to making sure the oracles are accurate, some issues are always present: actual typos (e.g. "buf" → "bufer"); ambiguous splits – different programmers split the same identifier in a different ways (e.g., "invalid-username" → {*invalid*, *user*, *name*} versus {*invalid*, *username*}); semantically equivalent splits but syntactically different; linguistic issues (e.g., "reparse" is often considered a unique term but it is not an English word) for example. Regarding the oracles that contain the exact set of terms, i.e., abbreviations are expanded, there are some issues with the exact expansion chosen (e.g., "auth" → "authenticate" versus "authentication"), or lack of consistency, i.e., abbreviations lacking an expansion, just to make it clear abbreviations considered acronyms (e.g., HTML, XML, SQL) are usually not expanded in the oracles. A human looking at the resulting split for each identifiers can cope with most of these issues, but all the processes that compute metrics over the resulting splits are done automatically, introducing some measurement errors.

The extra data files (including results) required for both experimental comparisons discussed in this section are available online from http://conclave.di.uminho.pt/articles/.

6. Conclusion

This paper introduces LIDS, a technique for bringing program identifiers from the program domain, to the natural language domain, this means converting program identifiers to sets of full natural language (including domain specific) terms. This involves expanding abbreviations and splitting multi-term identifiers for any combination of techniques and shortcuts used by programmers. This usually increases the semantic value of identifiers to enhance software analysis tool outcomes.

The described approach is in line with current state-of-the-art research. The use of NLP techniques to create linguistic artifacts, like dictionaries and taxonomies, has contributed with domain context vocabulary for example, improving the accuracy of the tokenization process, and the expansion of abbreviations.

Besides the main algorithm to compute splits, this work also proposes a technique to produce software domain specific dictionaries, automatically calculated from the documentation corpus. These dictionaries provide valuable information about domain specific terms and expressions that are prone to be used to compose identifiers in the software domain, but do not belong to general programming dictionaries. Another benefit of having a specific vocabulary domain is to correctly expand abbreviations that can have different expansions depending on context.

The presented approach was applied to identifier sets extracted from two software packages written in C, using an heterogeneous set of methods to create multi-word identifiers and abbreviations. The computed sets were then compared to the manually created sets of splits and expanded terms. The empirical study results show that this technique outperforms an explicit *hard* split approach, reaching a precision of 95% for correct splits, and 91% for correct terms, for the *AbcMidi* package, using the custom corpus-based dictionary. Results achieved for the *tree* package are less good mainly because of the package size, providing less domain specific text – smaller corpus. LIDS was also compared with other state-of-the-art techniques available, achieving results in the same range of values. The results achieved for the *JHotDraw* and *Lynx* packages although are not the best, are close to other state-of-the-art approaches. The results achieved for the LUDISO oracle are lower mainly because of the lack of the custom corpus-based dictionary for the various software packages.

Of course we are aware that more tests should be performed to provide a fair comparison and take solid conclusions about LIDS performance. This is endorsed for future work but will depend on the availability of other tools and approaches, including related resources (e.g., dictionaries) to conduct the necessary experiments. Along the paper we raised up four research questions that were discussed and answered. To conclude the paper it is important to enhance our contributions. On one hand, we have shown that the tool we developed for identifiers extraction, splitting and expansion – LIDS – has a performance in line with current state-of-the-art approaches that, to the best of our knowledge, have been published. On the other hand, we also showed that LIDS has a performance boost in very specific domains due to the fact that it is supported by a domain-specific dictionary that is build automatically from the software documentation available; results attained in our experiments are very promising in this respect. We believe that this facet can be further explored by other researchers in different contexts to improve even more the tool accuracy. But, for sure, the main contribution is the easy way how LIDS can be downloaded and used. It is now available from CPAN, can be installed in the standard ways, and can be used as a stand-alone tool or integrated as a library in other systems. The new algorithm proposed, that resort to different and customizable dictionaries, and the simple way how it can be integrated are the stronger reasons that led us to think that LIDS is actually an alternative to similar systems available. Also, this tool can be used to split multi-term identifiers in other contexts, outside the scope of software engineering.

Future work involves applying this technique to more software packages in different domains, a more solid comparison with other state-of-the-art approaches, and take in account context information about identifiers if available. It is also planned the contribution for a *splitting tokenizing benchmark suite* – a set of shared software packages, identifiers sets and corresponding *oracles* – to allow more accurate comparative studies. Further future work, includes the integration of this technique (and similar ones) with a broader scope system for software analysis, to create semantically rich ontological representations of programs. Which can be linked with the problem ontology to enhance the creation of mappings between software elements and real world concepts.

Acknowledgments

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.

We would like to thank the reviewers for their valuable insight and detailed comments, which aided in improving this paper.

We would like to thank Latifa Guerrouj, Philippe Galinier, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Massimiliano Di Penta, for their work in Guerrouj et al. (2012), and Emily Hill, David Binkley, Dawn Lawrie, Lori Pollok and K. Vijay-Shanker for their work in Hill et al. (2013), which allowed the experimental comparison between approaches.

References

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

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E., 2002. Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. 28 (10), 970–983.
Butler, S., Wermelinger, M., Yu, Y., Sharp, H.,July 2011. Improving the tokenisation of identifier names. In: 25th European Conference on Object-Oriented Programming. Springer.
Caprile, B., Tonella, P.,1999. Nomen est omen: Analyzing the language of function identifiers. In: Proceedings. Sixth Working Conference on Reverse Engineering. IEEE, pp. 112–122.
Caprile, B., Tonella, P.,2000. Restructuring program identifier names. In: Proceedings. International Conference on Software Maintenance. IEEE, pp. 97–107.
Carvalho, N.R., Almeida, J.J., Pereira, M.J.V., Henriques, P.R., 2012. Probabilistic synset based concept location. In: SLATE'12 – Symposium on Languages, Applications and Technologies, pp. 239–253, http://dx.doi.org/10.4320/OASIcs.SLATE.2012.I.
Carvalho, N.R., Simões, A., Almeida, J.J., 2013. Open source software documentation mining for quality assessment. In: Advances in Information Systems and Technologies. Springer, pp. 785–794.
Carvalho, N.R., Simões, A., Almeida, J.J., 2014. Dmoss: open source software dcumentation assessment. Comput. Sci. Inform. Syst. (submitted for publication; http://www.doiserbia.nb.rs/Article.aspx?id=1820-02141400027CAspxAuto DetectCookieSupport=1#.VFja3IusU00).
Chikofsky, E., Cross II, J., 1990. Reverse engineering and design recovery: a taxonomy. IEEE Softw., 13–17.
Corazza, A., Di Martino, S., Maggio, V.,2012. Linsen An efficient approach to split identifiers and expand abbreviations. In: 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp. 233–242.
Corbi, T., 1989. Program understanding: challenge for the 1990. IBM Syst. J. 28 (2), 294–306.
Dit, B., Guerrouj, L., Poshyvanyk, D., Antoniol, G.,2011. Can better identifier splitting techniques help feature location? In: IEEE 19th International Conference on Program Comprehension (ICPC). IEEE, pp. 11–20.
Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D., 2013. Feature location in source code: a taxonomy and survey. J. Softw.: Evol. Process 25 (1), 53–95, http://dx.doi.org/10.1002/smr.567.
Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K.,2009. Mining source code to automatically split identifiers for software analysis. In: MSR'09. 6th IEEE International Working Conference on Mining Software Repositories. IEEE, pp. 71–80.
Feild, H., Binkley, D., Lawrie, D., 2006. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In: Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06), Citeseer.
Guerrouj, L., Galinier, P., Gueheneuc, Y., Antoniol, G., Di Penta, M.,2012. Tris: a fast and accurate identifiers splitting and expansion algorithm. In: 19th Working Conference on Reverse Engineering (WCRE). IEEE, pp. 103–112.
Guerrouj, L., Di Penta, M., Antoniol, G., Guéhéneuc, Y.-G., 2011. Tidier: an identifier splitting approach using speech recognition techniques. J. Softw. Maint. Evol.: Res. Pract. 25 (6), 575–599.
Halliday, M.A., 1992. Language as system and language as instance: the corpus as a theoretical construct. Dir. Corpus Linguist., 61–77.
Hill, E., Binkley, D., Lawrie, D., Pollock, L., Vijay-Shanker, K., 2013. An empirical study of identifier splitting techniques. Empir. Softw. Eng., 1–27.
Kennedy, G. Introduction to Corpus Linguistics, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.75.8645
Lawrie, D., Binkley, D.,2011. Expanding identifiers to normalize source code vocabulary. In: 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp. 113–122.
Lawrie, D., Morrell, C., Feild, H., Binkley, D., 2006. What's in a name? A study of identifiers. In: 14th International Conference on Program Comprehension, Citeseer.
Lawrie, D., Feild, H., Binkley, D.,2006. Syntactic identifier conciseness and consistency. In: SCAM'06. Sixth IEEE International Workshop on Source Code Analysis and Manipulation. IEEE, pp. 139–148.
Lawrie, D., Morrell, C., Feild, H., Binkley, D., 2007. Effective identifier names for comprehension and memory. Innov. Syst. Softw. Eng. 3 (4), 303–318.
Lawrie, D., Feild, H., Binkley, D.,2007. Extracting meaning from abbreviated identifiers. In: SCAM 2007. Seventh IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 213–222.
Lawrie, D., Binkley, D., Morrell, C.,2010. Normalizing source code vocabulary. In: 17th Working Conference on Reverse Engineering (WCRE). IEEE, pp. 3–12.
Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.,2007. Feature location via information retrieval based filtering of a single scenario execution trace. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 234–243.
Madani, N., Guerrouj, L., Di Penta, M., Gueheneuc, Y., Antoniol, G.,2010. Recognizing words from source code identifiers using speech recognition techniques. In: 14th European Conference on Software Maintenance and Reengineering (CSMR). IEEE, pp. 68–77.
Marcus, A., Maletic, J.I.,2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings. 25th International Conference on Software Engineering. IEEE, pp. 125–135.
Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.,2004. An information retrieval approach to concept location in source code. In: Proceedings. 11th Working Conference on Reverse Engineering. IEEE, pp. 214–223.
Martin, J.H., Jurafsky, D., 2000. Speech and Language Processing.
Nelson, M. A Survey of Reverse Engineering and Program Comprehension, http://arxiv.org/abs/cs/0503068
Rajlich, V., Wilde, N.,2002. The role of concepts in program comprehension. In: Proceedings. 10th International Workshop on Program Comprehension. IEEE, pp. 271–278.
Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K., 2007. Using natural language program analysis to find and understand action-oriented concerns. In: Int. Conf, on Aspect-oriented Software Development.
Sureka, A.,2012. Source code identifier splitting using yahoo image and web search engine. In: Proceedings of the First International Workshop on Software Mining. ACM, pp. 1–8.
Von Mayrhauser, A., Vans, A., 1995. Program comprehension during software maintenance and evolution. Computer 28 (8), 44–55.
Yadla, S., Hayes, J.H., Dekhtyar, A., 2005. Tracing requirements to defect reports: an application of information retrieval techniques. Innov. Syst. Softw. Eng. 1 (2), 116–124.

Nuno Ramos Carvalho is currently a PhD student in University of Minho in the MAP-i doctoral program. His main areas of research interests are conception and implementation of domain specific languages, specification of languages, automatic construction of compilers and other language-based tools, and software engineering and reverse engineering. His thesis is in the area of program comprehension, more specifically trying to understand how a program works without any previous knowledge about it's implementation or design goals, by relating information retrieved from the program and problem domain.

José João Almeida teaches in department of informatics, University of Minho in the area of compilers and natural language processing.

Pedro Rangel Henriques got a degree in "Electrotechnical/Electronics Engineering", at FEUP (Porto University), and finished a Ph.D. thesis in "Formal Languages and Attribute Grammars" at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the "Language Processing group" at CCTC (Computer Science and Technologies Center). He teaches many different courses in the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visulaization/animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He is co-author of the "XML & XSL: da teoria a prática" book, published by FCA in 2002, has published 6 chapters in books, and 26 journal papers, and has been enrolled in 28 R&D projects.

Maria João Varanda received the M.Sc. and Ph.D. degrees in Computer Science from the University of Minho in 1996 and 2003 respectively. She is a member of the Language Processing group in the Computer Science and Technology Center, at the University of Minho. She is currently an adjunct professor at the Technology and Management School of the Polytechnic Institute of Bragança, on the Informatics and Communications Department and vice-president of the same school. As a researcher of gEPL, she is working with the development of compilers based on automatic generation tools, visual languages, domain specific languages and program comprehension. She is author or coauthor of 13 journal papers and over 47 international conference papers.