2010 14th European Conference on Software Maintenance and Reengineering

Recognizing Words from Source Code Identifiers using Speech Recognition Techniques

Nioosha Madani*, Latifa Guerrouj*, Massimiliano Di Penta†, Yann-Gaël Guéhéneuc‡, Giuliano Antoniol*
**SOCCER Lab. – DGIGL, École Polytechnique de Montréal, Québec, Canada*
†*University of Sannio - Department of Engineering - Benevento - Italy*
‡*Ptidej Team – DGIGL, École Polytechnique de Montréal, Québec, Canada*
{nioosha.madani,latifa.guerrouj,yann-gael.gueheneuc}@polymtl.ca,
dipenta@unisannio.it, antoniol@ieee.org

***Abstract*—The existing software engineering literature has empirically shown that a proper choice of identifiers influences software understandability and maintainability. Researchers have noticed that identifiers are one of the most important source of information about program entities and that the semantic of identifiers guide the cognitive process.**

Recognizing the words forming identifiers is not an easy task when naming conventions (*e.g.*, Camel Case) are not used or strictly followed and–or when these words have been abbreviated or otherwise transformed. This paper proposes a technique inspired from speech recognition, *i.e.*, dynamic time warping, to split identifiers into component words.

The proposed technique has been applied to identifiers extracted from two different applications: JHotDraw and Lynx. Results compared to manually-built oracles and with Camel Case algorithm are encouraging. In fact, they show that the technique successfully recognizes words composing identifiers (even when abbreviated) in about 90% of cases and that it performs better than Camel Case. Furthermore, it was able to spot mistakes in the manually-built oracle.

***Keywords*—Source code identifiers; program comprehension.**

I. Introduction

One of the problems that developers face when understanding and maintaining a software system is that, very often, documentation is scarce, outdated, or simply not available. This problem is not limited to open source projects but it is also true in industry: as systems evolve, documentation is not updated due to time pressure and the need to reduce costs. Consequently, the only up-to-date source of information is the source code and therefore identifiers and comments are key means to support developers during their understanding and maintenance activities.

In their seminal work, Deißenböck and Pizka highlighted that proper identifiers improve quality and that *"it is the semantics inherent to words that determine the comprehension process"* [1]. Indeed, according to Tonella and Caprile [2], [3], the identifiers are one of the most important source of information about system concepts.

Researchers have studied the usefulness of identifiers to recover traceability links [4], [5], [6], measure conceptual cohesion and coupling [7], [8], and, in general, as assets that can highly affect source code understandability and maintainability [9], [10], [11]. Researchers have also studied the quality of source code comments and the use of comments and identifiers by developers during understanding and maintenance activities [11], [12], [13]. They all concluded that identifiers can be useful if carefully chosen to reflect the semantics and role of the named entities.

In the following, we will refer to any substring in a compound identifier as a *term*, while an entry in a dictionary (*e.g.*, the English dictionary) will be referred to as a *word*. A term may or may not be a dictionary word. A term carries a single meaning in the context where it is used, while a word may have multiple meanings (upper ontologies like WordNet[1] associate multiple meanings to words).

Stemming from Deißenböck and Pizka's observation on the relevance of words and terms in identifiers to drive program comprehension several previous works attempted to segment identifiers by splitting them into component terms and words.

Indeed, identifiers are often composed of terms reflecting domain concepts [11], referred to as *"hard words"*. Hard words are usually concatenated to form compound identifiers, using the Camel Case naming convention, *e.g.*, *drawRectangle*, or underscore, *e.g.*, *draw_rectangle*. Sometimes, no Camel Case convention or other separator (*e.g.*, underscore) is used. Also, acronyms and abbreviations may be part of any identifier, *e.g.*, *drawrect* or *pntrapplicationgid*. The component words *draw, application*, the abbreviations *rect, pntr*, and the acronym *gid* (*i.e.*, group identifier) are referred to as *"soft-words"* [14].

To the best of our knowledge, two families of approaches exist to split compound identifiers: the simplest one assumes the use of the Camel Case naming convention or the presence of an explicit separator. The more complex strategy is implemented by the *Samurai* tool and it relies on a lexicon and uses greedy algorithms to identify component words. *Samurai* [15] is a technique and a tool that assumes that an identifier is composed of words used (alone) in some

[1] *http://wordnet.princeton.edu*

1534-5351/10 $26.00 © 2010 IEEE
DOI 10.1109/CSMR.2010.31

other parts of the system. It therefore uses words and word frequencies, mined from the source code, to determine likely splittings of identifiers.

However, the above mentioned approaches have limitations. First, they are not always able to associate identifier substrings to words or terms, *e.g.*, domain-specific terms or English words, which could be useful to understand the extent to which the source code terms reflect terms in high-level artifacts [16]. Second, they do not deal with word transformations, *e.g.*, abbreviation of *pointer* into *pntr*.

This paper proposes a novel approach to segment identifiers into composing words and terms. The approach is based on a modified version of the Dynamic Time Warping (DTW) algorithm proposed by Ney for connected speech recognition [17] (*i.e.*, for recognizing sequences of words in a speech signal) and on the Levenshtein string edit-distance [18]. The approach further assumes that there is a limited set of (implicit and–or explicit) rules applied by developers to create identifiers. It uses words transformation rules, plus a hill climbing algorithm [19] to deal with word abbreviation and transformation.

The main contributions of this paper are the following:

1) A new approach to split identifiers, inspired from speech recognition. The approach overcomes limitations of previous approaches and can split identifiers composed of transformed words, regardless of the kind of separators;
2) Evidence that the approach can be used to map transformed words composing identifiers to dictionary words and, therefore, to build a thesaurus of the identifiers;
3) Results of applying our approach two software systems belonging to different domains, JHotDraw (written in Java) and Lynx (written in C). Results based on manually-built oracles indicate that the approach can correctly split up to 96% of the identifiers and can even be used to identify errors in the oracle.

This paper is organized as follows. Section II describes the novel approach to split identifiers, also reporting a primer on Ney's connected-words recognition algorithm [17]. Section III reports an empirical study aimed at evaluating the proposed approach. Section IV relates this work to the existing literature. Finally, Section V concludes the paper and outlines future work.

II. The Identifier Splitting Approach

Often programmers build new identifiers by applying a set of transformation rules to words, such as dropping all vowels (*e.g.*, *pointer* becomes *pntr*), or dropping one or more characters (*e.g.*, *pntr* becomes *ptr*).

Our goal is to provide a meaning to simple and composed identifiers, even in presence of such truncated/abbreviated words, *e.g.*, *objectPrt*, *cntr* or *drawrect*, by associating identifier (substrings) to terms and words; words belonging to either a full English dictionary or to a domain-specific or an application-specific dictionary.

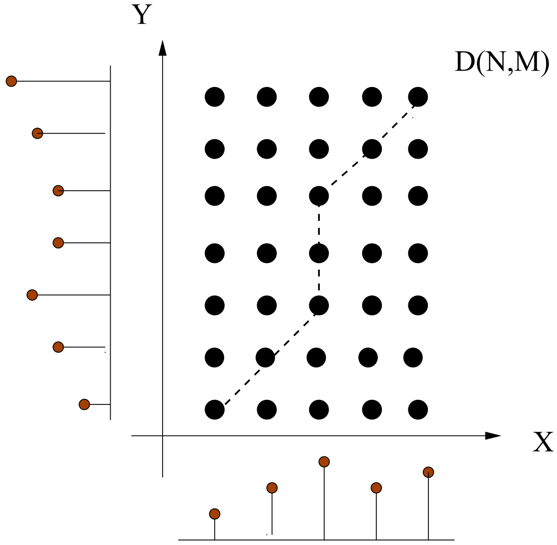

Figure 1. Dynamic Time Warping (DTW) of two time signals.

This section describes the approach for identifying terms and words in source code identifiers. The approach takes as input two artifacts (i) the set of identifiers to be segmented into words and terms and (ii) a dictionary containing words and terms belonging to an upper ontology, to the application domain, or both.

Examples of dictionaries that can be used for our purpose are, for example, WordNet (which contains around 90,000 entries) or dictionaries used by spell checkers, such as a-spell (which contains around 35,000 English words in a typical configuration). Each dictionary word may be associated to a set of known abbreviations in a way similar to a thesaurus. For example, the *pointer* entry in the dictionary can be associated to abbreviations *pntr, ptr* found as terms composing identifiers. Thus, if *pntr* is matched, the algorithm can expand it into the dictionary term *pointer*. The overall idea is to identify near optimal matching between substrings in identifiers and words in the dictionary, using an approach inspired by speech recognition.

A. Dynamic Programming Algorithm for Identifier Splitting

Identifier splitting is performed via an adaptation of the connected speech recognition algorithm proposed by Ney [17] that, in turns, extends to connected words the isolated word DTW [20] algorithm. DTW was conceived for time series alignment and was widely applied in early speech recognition applications in the 70s and 80s.

As shown in Figure 1, given two signals, two vectors of features, *i.e.*, coefficients extracted from two utterances, the algorithm determines an alignment between the two vectors of features.

Let $x_1, x_2, \ldots, x_N$ be the input feature vector extracted from an unknown utterance (x axis) and $y_1, y_2, \ldots, y_M$ a feature vector in a dictionary of signals (y axis) a DTW algorithm performs a warping of the time axis x and y to find the optimal match between the two vectors and, thus, the closest match of the unknown input utterance with prerecorded dictionary entries.

To compute the optimal matching given two time series (or strings) $x_1, x_2, \ldots, x_N$ and $y_1, y_2, \ldots, y_M$, the DTW distance $D(N, M)$ is recursively computed. Let $d(x_i, y_j)$ be a local distance chosen depending on the problem at hand. In speech recognition, such a distance is often the Euclidean distance between feature vectors; for strings, it can be the ordinal distance of characters or just the comparison of the two characters at position i and j [21]. Then, for any intermediate point $D(i, j)$:

$$\begin{aligned} c(i,j) &= d(x_i, y_j) \\ D(i,j) &= min[w_1 \cdot D(i-1,j) + c(i,j) \quad , \quad // \; insertion \\ & \quad w_2 \cdot D(i,j-1) + c(i,j) \quad , \quad // \; deletion \\ & \quad D(i-1,j-1) + w_3 c(i,j)] \quad // \; match \end{aligned}$$

The recurring equation computes the current distance based on previous values and thus it imposes continuity constraints. Weights w_1, w_2, w_3 are problem-dependent, typically w_1 is chosen equal to w_2, so that the computed distance is symmetric; w_3 is often chosen to be twice the value of w_1 and w_2. In string matching, if x_i differs from y_j, then it corresponds to a substitution that is equivalent to the deletion of one character followed by one insertion. In our computation, we choose $w_1 = w_2 = 1$ and $w_3 = 2$, as in the classic Levenshtein string edit distance [18].

The computation is done as follows. It uses a grid built by putting an identifier on the x-axis and some dictionary words on the y-axis, as shown in Figure 2. It starts on the bottom-left side of the grid and is performed by computing a distance $D(i, j)$ based on $d(x_i, y_j)$ for each cell of the grid, *i.e.*, by comparing the corresponding elements on the x and y axis (which are portions of the signal in speech recognition, while they are characters in our application) and finding the local path of minimum cost, *i.e.*, $(i-1, j)$, $(i, j-1)$, or $(i-1, j-1)$. The computation proceeds by columns (or rows); once the cost matrix $D(i, j)$ has been filled, the cell $D(N, M)$ contains the minimum alignment cost, *i.e.*, the minimum distance between $x_1, x_2, \ldots, x_N$ and $y_1, y_2, \ldots, y_M$. Backtracking from (N, M) down to $(0, 0)$ recovers the warping path corresponding to the optimal alignment of $x_1, x_2, \ldots, x_N$ and $y_1, y_2, \ldots, y_M$.

Given a dictionary containing the words rotate and shape, the identifier *rotateShape* would best match with the words *rotate* and *shape*. The identifier splitting problem can thus be brought back to the problem of determining the sequence of R words $q(1), \ldots, q(R)$, where $q(i)$ represents a word in a given dictionary, such that the distance between the input identifier and the sequence of word is minimized. Figure 2 graphically represents this problem, where the x axis represents an identifier (composed of one or more words) and on y axis are entries of a reference dictionary. In Figure 2, the dictionary contains three words *Counter*, *User*, and *Ptr*, while the input identifier is *UserCounterPtr* (our approach is case insensitive).

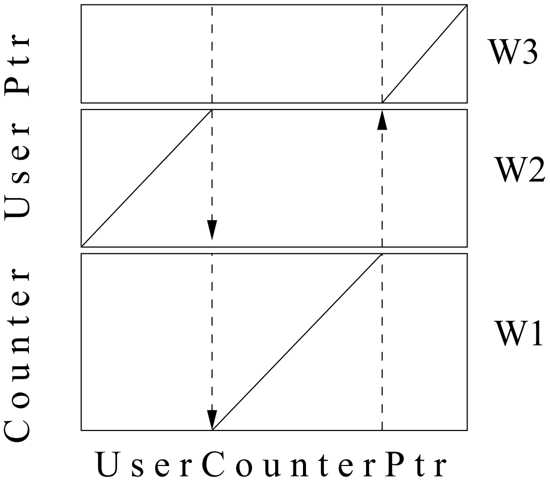

Figure 2. Connected word recognition (from [17]).

The warping problem can be thought as having multiple instances of the problem represented in Figure 1, one instance for each dictionary entry as shown in Figure 2, where at each column end, *e.g.*, column i, each word in the dictionary can start at the next position $i+1$. In other words, the algorithm performs a warping of each word and then identifies an optimal path among these warpings (represented by dashed arrows in Figure 2) to match the signal.

Further details about the matching algorithm can be found in [17]. The main difference between what is done in speech recognition and what we do in our context is in the used distance. In speech recognition, the distance adopted depends of the speech representation and it is usually an Euclidean, a Cepstral, or an Itakura–Saito distance. In our context, a comparison between characters of strings needs to be performed and thus $d(x_i, y_j)$ is the result of the comparison of characters x_i and y_j.

B. Word Transformation Rules

Some identifier substrings may not be part of the dictionary and need to be either generated from existing dictionary entries or added to it. Moreover, several words may have the same (minimum) distance from the substring to be matched when matching a substring of the identifier to the dictionary words to determine which word has the minimum distance from the substring and, thus, must be chosen, as shown in Figure 2.

Let us consider the identifier *fileLen* and suppose that the dictionary contains the words *length*, *file*, *lender*, and *ladder*. Clearly, the word *file* matches with zero distance the first four characters of *fileLen*, while both *length* and *lender* have a distance of three from *len*, because their last three characters could be dropped. Finally, the distance of *ladder* to *len* is higher than that of other words because only *l* matches. Thus, both *length* and *lender* should be preferred over *ladder* to generate the missing dictionary entry *len*.

To choose the most suitable word to be transformed, we use the following simple heuristic. We select the closest words, with non-zero distance, to the substring to be matched and repeatedly transform them using transformation rules chosen randomly among six possible rules. This process continues until a transformed word matches the substring being compared or when transformed words reach a length shorter than or equal to three characters. The available transformation rules are the following:

- *Delete all vowels:* All vowels contained in the dictionary word are deleted, *e.g.*, *pointer* → *pntr*;
- *Delete Suffix:* suffixes—such as *ing*, *tion*, *ed*, *ment*, *able*—are removed from the word, *e.g.*, *improvement* → *improve*;
- *Keeping the first n characters only*: the word is transformed by keeping the first n characters only, *e.g.*, *rectangle* → *rect* for $n = 4$;
- *Delete a random vowel:* one randomly chosen vowel from the word is deleted, *e.g.*, *number* → *numbr*;
- *Delete a random character: i.e.*, one randomly-chosen character is omitted, *e.g.*, *pntr* → *ptr*.

The transformations are applied in the context of a hill-climbing search. DTW, word transformation rules, and hill climbing are the key components of our identifier segmentation algorithm, which works as follows:

1) Based on the current dictionary, we (i) split the identifier using DTW—as explained in Section II-A, (ii) compute the global minimum distance between the input identifier and all words contained in the dictionary, (iii) associate to each dictionary word a fitness value based on its distance computed in step (ii). If the minimum global distance in step (ii) is zero, the process terminates successfully; else
2) From dictionary entries with non-zero distance obtained at step (1), we randomly select one word having minimum distance and then:
    a) We randomly select one transformation not violating transformation constraints, apply it to the word, and add the transformed word to a temporary dictionary;
    b) We split the identifier via DTW and the temporary dictionary and compute the minimum global distance. If the added transformed word reduces the global distance, then we add it to the current dictionary and go to step (1); else
    c) If there are still applicable transformations, and the string produced in step (a) is longer then three characters, we go to step (a);
3) If the global distance is non-zero and the iteration limit was not reached, then, we go to step (1), otherwise we exit with failure.

These previous steps describe a hill climbing algorithm, in which a transformed word is added to the dictionary if and only if it reduces the global distance. Briefly, a hill climbing algorithm [19] searches for a (near) optimal solution of a problem by moving from the current solution to a randomly chosen, nearby one, and accepts this solution only if it improves the problem fitness (the distance in our case). The algorithm terminates when there is no moves to nearby solutions improving the fitness. Differently from traditional hill climbing algorithms, in steps (a) and (b), our algorithm attempts to explore as much as possible of neighboring solutions by performing word transformations. Different neighbors can be explored depending on the order of transformations.

C. (Dis)Advantages of Applying DTW for Identifier Splitting

The usage of techniques inspired from speech recognition is not the only way of splitting identifiers into words. Clearly, when Camel Case separators (or other separators, such as the underscore) are being used, there is no need for complex splitting techniques. However:

- In some situations, the Camel Case separator or other explicit separators are not used, thus other approaches must be used. A possible alternative approach is the one by Enslen *et al.* [15];
- The DTW algorithm is able to provide a distance between an identifier and a set of words in a dictionary even if there is no perfect match between substrings in the identifier and dictionary words; for example, when identifiers are composed of abbreviations, *e.g.*, *getPntr*, *filelen*, or *DrawRect*. It accepts a match by identifying the dictionary words closest to identifier substrings;
- The DTW algorithm is able to perform an alignment when matching words from the dictionary, thus it is able to work even when the word to be matched is preceded or followed by other characters, *e.g.*, *xpntr*; thus, it is better than, for example, applying only the Levenshtein edit distance.
- The DTW algorithm assigns a distance to matched substrings. Thus, in the above *fileLen* example, we would discover that *file* matches the first four characters with a zero distance (thus $distance = 0$) and that *length* matches the five to seven characters (at $distance = 3$);

- The dictionary can be sorted so that the approach favors the matching of the longest words, with respect to multiple words composing the longest one. Thus, the identifier *copyright* would be matched to the word *copyright* rather than to the composition of words *copy* and *right*, which also belong to the dictionary.

DTW is a powerful technique, however it has also some disadvantages. The first disadvantage is the intrinsic quadratic complexity of a single match with a cubic cost when we consider a dictionary. Furthermore, sentence syntax and semantics are not involved as matching is done at the character level. Going back to the *fileLen* example, *length* should be preferred over *lender*, however DTW cannot choose between the two. Finally, developers are able to disambiguate complex situations leading to optimal non-zero distance split when DTW cannot. Consider the identifier *imagEdges*; it is immediate to recognize the component words *image* and *edges*. However, *image* and *edges* match the identifier with a distance of 1 because the *E* character is shared by both terms in the identifier and, thus, the optimal minimum cost is 1 and not 0.

Our approach deals with similar disadvantages by transforming words and running multiple times the DTW algorithm to build multiple candidate splittings. Clearly, any developer would use syntax and semantics as well as her knowledge of the domain and context implicitly: even if *imag* is not a well-formed English word, she will correctly split *imagEdges* into *image* and *edges*.

III. Empirical Study

The *goal* of this study is to analyze the proposed identifier splitting approach, with the *purpose* of evaluating its ability to adequately identify dictionary words composing identifiers, even in presence of word transformations. The *quality focus* is the precision and recall of the approach when identifying words composing the identifiers with respect to a manually-built oracles. The *perspective* is of researchers, who want to evaluate an approach for identifier splitting, that can be used as a means to assess the quality of source code identifiers, *i.e.*, the extent to which they would refer to domain words or in general to meaningful words, *e.g.*, words belonging to a requirements' dictionary.

The *context* consists of a dictionary and identifiers extracted from the source code of two software systems, JHotDraw and Lynx. The dictionary contains about 2,500 words extracted from a glossary found on the Internet[2], 500 most frequent English words[3], plus terms and words contained in Lynx and JHotDraw.

JHotDraw[4] is a Java framework for drawing 2D graphics. The project started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. *Lynx*[5] is known as "the textual Web browser", *i.e.*,, it is a free, open-source, text-only Web browser and Gopher client for use on cursor-addressable, character cell terminals. Lynx is entirely written in C. Its development began in 1992 and it is now available on several platforms, including Linux, UNIX, and Windows. Table I reports some relevant figures about the two systems that we analyzed.

Table I
Main characteristics of the two analyzed systems.

Metrics	JHotDraw	Lynx
Analyzed Releases	5.1	2.8.5
Files	155	247
KLOCs	16	174
Identifiers (> 2 chars)	2,348	12,194

A. Research Questions

The study reported in this section aims at addressing the following research questions:

1) **RQ1:** *What is the percentage of identifiers correctly split by the proposed approach?* This research question investigates the overall performance of our approach, comparing the results with a manually-built oracle.
2) **RQ2:** *How does the proposed approach perform compared with the Camel Case splitter?* This research question compares the performance of the proposed approach with the simple Camel Case splitter, specifically the capability of correctly splitting identifiers and of mapping substrings to dictionary words.
3) **RQ3:** *What percentage of identifiers containing word abbreviations is the approach able to map to dictionary words?* This research question evaluates the ability of the proposed approach to map identifier substrings to dictionary words when these substrings represent abbreviations of dictionary words.

B. Analysis Method

The above research questions aim at understanding if the proposed approach helps in decomposing identifiers. Thus, we implicitly assume that, given an identifier, there exists an exact subdivision of this identifier into terms and words that, possibly after transformations and once concatenated, compose the identifier. First, we limited our analysis to identifiers longer than or equal to three characters: 2,348 in JHotDraw and 12,194 in Lynx. We have explicitly split identifiers containing digits, *e.g.*, *name4Tag* into *name* and *tag* and *sent2user* into *sent* and *user*, because our approach cannot map *2* to the word *to* and *4* to *for*, which are the intended meanings of these terms.

To evaluate our approach, we selected the 957 JHotDraw and 3,085 Lynx composed identifiers for which it was

[2] *http://www.matisse.net/files/glossary.html*

[3] *http://www.world-english.org/*

[4] *http://www.jhotdraw.org*

[5] *http://lynx.isc.org/*

possible to define a segmentation. We excluded from our analysis identifiers that were composed of one single English word, and identifiers for which it was not possible to clearly identify a splitting into dictionary words and an expansion of abbreviations. Examples of identifiers belonging to such a category are some identifiers extracted from Lynx source code, *e.g.*, *gieszczykiewicz*, *hmmm*, *ixoth*, *pqrstuvwxyz*, or *tiocgwinsz*. The 957 (3,085, respectively) identifiers were manually segmented into composing substring mapped into words and terms, thus, creating oracles for JHotDraw and for Lynx.

RQ1 aims at answering a preliminary research question about the applicability and usefulness of the proposed approach. To answer **RQ1**, we followed a two-steps approach. First, we executed the proposed algorithm in a single iteration mode and with no transformations. Thus, only identifiers composed of dictionary words are split with zero distance. Not-split identifiers, *i.e.*, with splitting distance not equal to zero, were fed into the second phase. In the second phase, we applied our approach with an upper bound of 20,000 iterations, *i.e.*, 20,000 dictionary word transformations and DTW splits. We chose 20,000 iterations as we noticed that after such a number of iterations, the approach was almost always able to find a splitting in a reasonable time, *i.e.*, within 2 minutes with our dictionary composed of 3,000 words. After automatic splitting have been performed, results are compared against the oracle, to compute the percentage of correctly segmented identifiers.

In phase two, we only included those identifiers that were not split in phase one and for which the composing substrings were longer than or equal to three characters, as shorter substrings were conservatively considered as spurious characters, pre-/post-fix or errors, thus penalizing our approach. Also, matching such short identifiers by performing transformations of dictionary words would not be feasible as too many dictionary words, after a sequence of transformations, would match the (short) substrings. For example, in the identifier *fpointer* the character *f* can be generated by any dictionary words containing the letter *f*. Much in the same way, the substring *ly* in Lynx identifiers such as *lysize* can be expanded to several different words.

RQ2 aims at performing a comparison of the proposed approach with the Camel Case splitting approach. We implemented a basic Camel Case identifier splitting algorithm and compared its results with the manually-built oracle. To statistically compare percentage of correct splittings performed by the proposed approach with those of the Camel Case splitter, we use Fisher's exact test [22] and tested the null hypothesis H_0: *the proportions of correct splittings obtained by the two approaches are not significantly different*.

To quantify the effect size of the difference between the two approaches, we also computed the *odds ratio* (OR) [22] indicating the likelihood of an event to occur, defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the percentage of identifiers correctly split by our approach (experimental group), to the odds q of it occurring in the other sample, *i.e.*, the percentage of identifiers correctly split by the Camel Case splitter (control group): OR $= \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 indicates that the event is equally likely in both samples. OR > 1 indicates that the event is more likely in the first sample (proposed approach) while an OR < 1 indicates the opposite (Camel Case splitter).

Table II
PERCENTAGE OF CORRECT CLASSIFICATIONS (RQ1).

Systems	Identifiers	Exact Splittings		Errors
		Single Iteration	Multiple Iterations	
JHotDraw	957	891 (93%)	920 (96%)	37
Lynx	3,085	2,169 (70%)	2,901 (94%)	217

RQ3 aims at assessing the ability of our approach to find identifiers splittings when component substrings are obtained by means of dictionary word transformations, such as in *rectpntr* using *pntr* instead of *pointer* and *rect* in place of *rectangle*. **RQ3** is addressed similarly to **RQ1**, comparing identifiers matched in phase two (as explained for **RQ1**) with the subset of the identifiers in the oracle that, according to our manual classification, contained abbreviations.

C. Study Results

This section reports results of the empirical study with the objective of addressing our research questions.

1) RQ1: What is the percentage of identifiers correctly split by the proposed approach?: Table II reports for JHotDraw and Lynx the results of the identifier splittings obtained with our approach. In particular, the third column reports the number of identifiers exactly split in a single step, *i.e.*, with DTW distance zero and matching the oracle. Results indicate that, for both systems, a large percentage of identifiers have been created via simple concatenations of dictionary words. In fact, 93% of JHotDraw identifiers, and 70% of Lynx identifiers have been exactly split into dictionary words within a single iteration of our algorithm.

The fourth column cumulates results of the third columns with the number of composed identifiers made of dictionary words abbreviations split with zero distance within 20,000 iterations. In other words, the fourth columns shows the numbers and percentages of all the correctly-split identifiers. Finally, the fifth column shows the number of identifiers that were not exactly split or for which the splitting did not match the oracle.

Wrong splittings were due to identifiers containing acronyms or short abbreviations. For example, we believe that it is impossible to identify correctly component words of the acronyms such as *afaik*, *imho*, or *foobar*. For different reasons, we also believe that it is impossible to find the exact splittings of identifiers such as *fsize*; even if we consider that the context of the identifiers *fsize* could be reasonably associated with both concepts of *file size* and *figure size*

Table III
PERFORMANCE OF THE CAMEL CASE SPLITTER.

Systems	Correct Splitings	Errors
JHotDraw	874	83
Lynx	561	2,524

depending on the JHotDraw code region where it is used, even though the letter *f* really means that the field is private.

Overall, about 96% of JHotDraw identifiers and 93% of Lynx identifiers were correctly segmented with zero distance. These results support our claim and conclusion that a very large fraction (above 90%) of identifiers can be exactly split by using our approach (**RQ1**).

2) RQ2: How does our approach compares to the Camel Case splitter?: Table III summarizes results of Camel Case splitting. Not surprisingly, the Camel Case approach works well on JHotDraw. Indeed, Java coding guidelines and identifier construction rules tend to promote Camel Case splitting and JHotDraw developers carefully followed coding standards and identifier creation rules. As the second line of the same table shows, this is not the case of Lynx, the C Web browser. Indeed, C coding standards such as the Indian Hill[6] coding standards or the GNU coding standards[7] do not enforce Camel casing.

When comparing the performances of the proposed approach (see Table II, considering results after the second phase, *i.e.*, the third column) with those of the Camel Case splitting (see Table III), the Fisher's exact test indicated no significant (or marginal) difference for JHotDraw (p-value $= 0.1$) with a OR $= 1.3$, *i.e.*, the proposed approach has chances of correctly splitting an identifier 1.3 more times than the Camel Case splitter. For Lynx, differences are statistically significant (p-value < 0.001) and we have an extremely high OR=60, *i.e.*, chances of our approach to correctly split identifiers are 60 times higher than the Camel Case splitter.

Therefore, we conclude that the proposed approach performs better than Camel Case splitter on both systems and significantly better on Lynx (**RQ2**).

3) RQ3: What percentage of identifiers containing word abbreviations is the approach able to map to dictionary words?: Table II and IV reports results aimed at addressing **RQ3**. The fourth and fifth columns of Table II show that a substantial fraction of identifiers containing abbreviations can be split into dictionary words that originate such abbreviations. More precisely, 44% and 70% of JHotDraw and Lynx identifiers containing abbreviations were correctly split into component words. The percentage of success for the two systems is quite different and the reason is the different ways in which identifiers have been composed. Indeed, in Lynx, very short prefixes are much more frequent and cryptic than in JHotDraw. In particular, Lynx prefixes, such as *ly*, *ht*, or *hta*, make it hard to produce correct splittings without a specialized dictionary in which such prefixes are added with, possibly, the proper expansion.

D. Discussion

The proposed approach has a non-deterministic aspect in the way word transformation rules are applied and in the way in which the candidate words to be transformed are selected. Consequently, different runs of the approach may lead to different identifier splittings. Table IV reports for a subset of JHotDraw identifiers the splittings obtained in ten runs, each run with an upper limit of 20,000 iterations. The lower part of the table shows identifiers wrongly segmented and for which the zero distance was never achieved. It also shows in the last column the splittings computed. Two phenomena can be observed. First, because the word *red* is part of the dictionary, the identifier *writeref* is split into *write red* with distance 1; 1 is also the minimum distance and, thus, *red* is always preferred over *reference*. This fact suggests the need for improving the heuristic to select the candidate word to be used in splitting an identifier as any word shorter than *reference* with the current simple heuristic (based on the matching distance) is preferred.

We believe that for words such as *selection***z***ordered,* **j***hotdraw,* *get***v***adjustable,* **f***imagewidth* and **f***imageheight*, it would be impossible to compute the correct splitting and identify originating words. For example, in our dictionary the character **f** is contained in about 300 words, each of these words could generate **f** in **f***imageheight*. We believe that a substantial reduction of the search space is needed to match single characters, for example, by coupling our algorithm with the approach of Enslen *et al.* [15], which would restrict the search to the dictionary words containing **f** to the words used in the same method, class, or package. Finally, *serialversion***uid** suffers of the problem of acronym expansion mentioned above. We believe that the dictionary should also be expanded with well-known acronyms, such as *afaik*, and with technical abbreviations, such as *uid*, *gid*, *smtp*. In general, the dictionary should contain as many words belonging to the application domain as possible. Requirement documents and user manuals are precious sources of such words.

The upper part of Table IV shows another limitation of the proposed approach. Sometimes, different component words are discovered in different runs. For example, the identifier *newlen* was split in two different ways: *new length* and *new lender*. Clearly, the latter splitting is semantically wrong: even if *lender* can generate *len*, in the (intended) context of *newlen*, the term *lender* is nonsensical. We believe that the heuristic choosing the words to be transformed needs to be improved, possibly by relying on the strategy derived from [15], *i.e.*, favoring words already used in the same context.

[6] *http://www.chris-lott.org/resources/cstyle/*
[7] *http://www.gnu.org/prep/standards/*

Table IV
JHOTDRAW: RESULTS AND STATISTICS FOR SELECTED IDENTIFIERS IN TEN SPLITS ATTEMPTS. 25%, 50% AND 75% INDICATE THE FIRST, SECOND (MEDIAN), AND THIRD QUARTILES OF THE RESULTS DISTRIBUTION RESPECTIVELY.

Identifiers	Successes	Min.	25 %	50 %	75 %	Max.	Split I	Split II
bord**dec**	yes	208	617	1,346	1,938	8,831	bord decimal	bord decision
anchor**len**	yes	154	689	1,220	3,097	7,056	anchor length	anchor lender
draw**rect**	yes	29	779	2,385	4,877	8,629	draw rectangle	
drawround**rect**	yes	77	6,509	10,300	17,403	19,173	draw round rectangle	
fill**rect**	yes	898	3,549	5,942	10,932	12,659	fill rectangle	
javadraw**app**	yes	86	480	972	4,582	6,965	java draw apply	java draw append
net**app**	yes	76	788	1,529	4,183	7,394	net apply	net append
new**len**	yes	176	534	600	704	2,503	new length	new lender
nothing**app**	yes	90	305	1,425	4,803	9,956	nothing apply	nothing application
addcolumn**info**	yes	457	1,296	1,806	2,631	4,146	add column information	add column inform
add**lbl**	yes	43	793	1,130	3,498	4,843	add label	
case**comp**	yes	124	327	437	938	1,836	case compare	case complete
serialversion**uid**	No						serial version did	
selection**z**ordered	No						selection ordered	
remove**fr**figurerequestremove	No						remove figure request remove	
jhotdraw	No						hot draw	
get**v**adjustable	No						get bad just able	
fimagewidth	No						him age width	
fimageheight	No						him age height	
write**ref**	No						write red	

Finally, it is important to remark that building an oracle for this kind of approach is a difficult and challenging task. Each composed identifier must be split in component words and abbreviations expanded into English words. We have experienced that the task is non trivial: we discovered eight mistakes in the initial JHotDraw oracle, while assessing our approach output and similar errors also occurred in the first version of the Lynx oracle (both oracles were fixed after such runs and the corrected ones were used to produce the results reported in this paper).

E. Threats to Validity

Threats to *construct validity* concern the relation between the theory and the observation. Here, this threat is mainly due to mistakes in the oracles. Indeed, we cannot exclude that errors are still present in the oracles, despite the corrections made and explained above. However, the discovered errors were less than 1% of the number of identifiers contained in the oracles, thus the presence of some errors would not greatly affect our results. Nevertheless, as the intent of the oracles is to explain identifiers semantics, we cannot exclude that a part of identifiers could have been split in different ways by the developers that originally created them. This problem is somehow related to guessing the developers' intent and we can only hope that, given the application domain, the class, file, method, or function containing the identifiers (and the general information that can be extracted from the source code and documentation), it will be possible to infer the developers' *likely* intent. We are working on improving our oracle and increasing the number of manually-split identifiers.

Threats to *internal validity* concern any confounding factor that could influence our results. In particular, these threats can be due to subjectiveness during the manual building of the oracles. We attempted to avoid any bias in the oracles by using the same oracles and simple string matching when comparing Camel Case splitter with our approach. Furthermore, both oracles were built by the same researcher and manually verified by other two people. Whenever a disagreement was detected, a majority vote was taken. The size of the oracle was chosen large enough to ensure that even an error of a few percent in splits would not have affected algorithm comparison.

Threats to *Conclusion validity* concern the relationship between the treatment and the outcome. Identifiers split exactly into dictionary words in a single iteration may sometime be split in a different way from the developers' intent. However, we do not claim any relation between the splitting produced and the semantics of the identifiers; this relation is left to the developers' judgment and experience. We limit ourselves to comparing our approach with the Camel Case splitter and validating the quality of computed splittings with respect to the oracles. Conclusion validity may play a role when we compared the effectiveness in detecting word abbreviations. To limit such a threat, we manually inspected all splittings produced with multiple iterations and word transformations.

Threats to *external validity* concern the possibility of generalizing our results. The study is limited to two systems: JHotDraw and Lynx. Yet, our approach is applicable to any other system. However, we cannot claim that similar results would be obtained with other systems. We have compared our approach with a Camel Case splitter but cannot be sure that their relative performances would remain the same on different systems. However, the two systems correspond to different domains and applications, have different sizes, are developed by different teams, with different programming languages. We believe this choice mitigates the threats to the external validity of our study.

IV. Related Work

The paramount role of program identifiers in program understanding, traceability recovery, feature and concept location tasks motivate the large body of relevant work in this area. In the following, we focus on the most relevant contributions given the focus of our research.

Identifier and program understanding has been investigated in [1], [2], [23], [24]. Other work [3], [4], [11], [25] attempted to investigate the information carried by the words composing identifiers, their syntactic structure and quality. The existence of *"hard words"* that encode core concepts into identifiers was the main outcome of the study by Anquetil *et al.* [23]. An in-depth analysis of the internal identifier structure was conducted by Caprile *et al.* [2], while guidelines for the production of high-quality identifiers have been provided by Deißenböck *et al.* [1]. Methods related to identifier refactoring were proposed by Caprile *et al.* [3] and Demeyer *et al.* [26]. Lawrie *et al.* [11] performed an empirical study to assess the quality of source code identifiers. The results of their study with over 100 programmers indicate that full words as well as recognizable abbreviations lead to better comprehension.

Some studies [4], [5], [6] report how identifiers can be used to recover traceability links. De Lucia *et al.* [16] proposed COCONUT, a tool highlighting to developers the similarity between source code identifiers and comments and words in high-level artifacts. They showed that this tool is helpful to improve the overall quality of identifiers and comments. Merlo *et al.* [25] analyzed informal information including identifiers and comments in programs.

The role of identifiers in mapping the domain model into the program model (*i.e.*, programming entities) was studied by Takang *et al.* [24]. A crucial role is recognized to the program lexicon and the coding standards in the so-called naturalization process of software immigrants [27].

Overall, the above previous work highlights the importance of properly-choosing identifiers for source code comprehensibility and maintainability. In this context, the application of our approach would be to map terms in source code identifiers to domain dictionary words to better assess the quality of these identifiers.

Many commonalities can be found with the work of Enslen *et al.* [15]. We share with them the goal of automatically splitting identifiers into component words. However, our approach is different. We do not assume the presence of Camel Casing nor of a set of known prefixes or suffixes. In addition, our approach automatically generates a thesaurus of abbreviations via transformations attempting to mimic the cognitive processes of developers when composing identifiers with abbreviated forms.

V. Conclusion

The proper choice of identifiers can help in promoting software understanding and thus software evolution. Often, identifiers are created by concatenating English terms and–or acronyms and abbreviated form of words identifying domain concepts. Recognizing terms composing identifiers is a non-trivial task when concatenation does not follow Camel Case rules or when abbreviations are used.

In this paper, we presented an algorithm inspired by Ney's extension of the Dynamic Time Warping (DTW) algorithm to split identifiers into component words. We coupled the DTW extension with transformation rules and hill climbing to infer a segmentation in identifiers composed of dictionary words and also of word abbreviations.

We applied our approach to split the identifiers of two systems, developed with different programming languages, and belonging to different application domains: JHotDraw and Lynx. Results have been evaluated comparing the obtained splittings with manually-built oracles. They showed that the proposed approach outperforms a simple Camel Case splitter. In particular, for Lynx, the Camel Case splitter was able to correctly split only about 18% of the identifiers versus 93% with our approach. On JHotDraw, the Camel Case splitter exhibited a correctness of 91% while our approach ensured 96% of correct results. Our approach was also able to map abbreviations to dictionary words, in 44% and 70% of cases for JHotDraw and Lynx, respectively.

Future work will be devoted to extend the evaluation of our approach to other systems and to introduce enhanced heuristics for term selection and word transformations, with the aim of improving the current performances.

VI. Acknowledgements

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution #950-202658) and by G. Antoniol Individual Discovery Grant.

References

[1] F. Deißenböck and M. Pizka, "Concise and consistent naming," in *Proc. of the International Workshop on Program Comprehension (IWPC)*, May 2005.

[2] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proc. of the Working Conference on Reverse Engineering (WCRE)*, Atlanta Georgia USA, October 1999, pp. 112–122.

[3] ——, "Restructuring program identifier names," in *Proc. of the International Conference on Software Maintenance (ICSM)*, 2000, pp. 97–107.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, pp. 970–983, Oct 2002.

[5] J. I. Maletic, G. Antoniol, J. Cleland-Huang, and J. H. Hayes, "3rd international workshop on traceability in emerging forms of software engineering (tefse 2005)." in *ASE*, 2005, p. 462.

[6] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing." in *Proceedings of the International Conference on Software Engineering*, 2003, pp. 125–137.

[7] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[8] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance*. Philadelphia, Pennsylvania, USA: IEEE CS Press, 2006, pp. 469 – 478.

[9] A. Takang, P. Grubb, and R. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[10] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.

[11] D. Lawrie, C. Morrel, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proc. of the International Conference on Program Comprehension (ICPC)*, 2006, pp. 3–12.

[12] B. Fluri, M. Würsch, and H. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007, pp. 70–79.

[13] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in PostgreSQL," in *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006*, 2006, pp. 179–180.

[14] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 27-29 September 2006, Philadelphia, Pennsylvania, USA*.

[15] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," *Mining Software Repositories, International Workshop on*, vol. 0, pp. 71–80, 2009.

[16] A. De Lucia, M. Di Penta, R. Oliveto, and F. Zurolo, "Improving comprehensibility of source code via traceability information: a controlled experiment," in *Proceedings of 14th IEEE International Conference on Program Comprehension*. Athens, Greece: IEEE CS Press, 2006, pp. 317–326.

[17] H. Ney, "The use of a one-stage dynamic programming algorithm for connected word recognition," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 32, no. 2, pp. 263–271, Apr 1984.

[18] V. I. Levenshtein, "Binary codes capable of correcting deletions,insertions, and reversals," *Cybernetics and Control Theory*, no. 10, pp. 707–710, 1966.

[19] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics - (2nd edition)*. Berlin Germany: Springer-Verlag, 2004.

[20] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 26, no. 1, pp. 43–49, Feb 1978.

[21] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research articles," *Softw. Test. Verif. Reliab.*, vol. 16, no. 3, pp. 175–203, 2006.

[22] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[23] N. Anquetil and T. Lethbridge, "Assessing the relevance of identifier names in a legacy software system," in *Proceedings of CASCON*, December 1998, pp. 213–222.

[24] A. Takang, P. Grubb, and R. Macredie, "The effects of comments and identifier names on program comprehensibility: An experimental study," *Journal of Programming Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[25] E. Merlo, I. McAdam, and R. De Mori, "Feed-forward and recurrent neural networks for source code informal information analysis," *Journal of Software Maintenance*, vol. 15, no. 4, pp. 205–244, 2003.

[26] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems Languages and Applications*. ACM Press, 2000, pp. 166–177.

[27] S. E. Sim and R. C. Holt, "The ramp-up problem in software projects: a case study of how software immigrants naturalize," in *ICSE '98: Proceedings of the 20th international conference on Software engineering*. Washington DC USA: IEEE Computer Society, 1998, pp. 361–370.