Research

# TIDIER: An Identifier Splitting Approach using Speech Recognition Techniques

Latifa Guerrouj[a], Massimiliano Di Penta[b], Giuliano Antoniol[a], and Yann-Gaël Guéhéneuc[a]

[a] *DGIGL / PTIDEJ Team / SOCCER Lab., École Polytechnique de Montréal, Québec, Canada*
[b] *RCOST-Dept. of Engineering, University of Sannio, Italy*

## SUMMARY

**The software engineering literature reports empirical evidence on the relation between various characteristics of a software system and its quality. Among other factors, recent studies have shown that a proper choice of identifiers influences understandability and maintainability. Indeed, identifiers are developers' main source of information and guide their cognitive processes during program comprehension when high-level documentation is scarce or outdated and when source code is not sufficiently commented.**

**This paper proposes a novel approach to recognize words composing source code identifiers. The approach is based on an adaptation of Dynamic Time Warping used to recognize words in continuous speech. The approach overcomes the limitations of existing identifier-splitting approaches when naming conventions (*e.g.*, Camel Case) are not used or when identifiers contain abbreviations.**

**We apply the approach on a sample of more than 1,000 identifiers extracted from 340 C programs and compare its results with a simple Camel Case splitter and with an implementation of an alternative identifier splitting approach, Samurai. The results indicate the capability of the novel approach: (i) to outperform the alternative ones, when using a dictionary augmented with domain knowledge or a contextual dictionary and (ii) to expand 48% of a set of selected abbreviations into dictionary words.**

**key words: Identifier Splitting, Program Comprehension, Linguistic Analysis**

[1] Correspondence to: Latifa Guerrouj – latifa.guerrouj@polymt.ca

## 1. Introduction

There is a general consensus among researchers [CT99, CT00, DP05, LFB06, EHPVS09] on the usefulness of identifier to improve software quality, program comprehension, and program understandability. Indeed, researchers have studied the usefulness of identifiers to recover traceability links [ACC$^+$02b, MACHH05, MM03], measure conceptual cohesion and coupling [MPF08a, PM06], and, in general, as an asset that can highly affect source code understandability and maintainability [TGM96, LMFB07, LMFB06]. Researchers have also studied the quality of source code comments and the use of comments and identifiers by developers during comprehension and maintenance activities [LMFB06, FWG07, JH06]. The general conclusion, anticipated in Deißenböck and Pizka's seminal work [DP05], is that proper identifiers improve quality and that "it is the semantics inherent to words that determine the comprehension process".

Identifiers are often composed by concatenating domain and jargon terms, *e.g.*, *hexadecimal socket*, acronyms, *e.g.*, *ssl*, abbreviations, *e.g.*, *win* for *windows* or *dir* for *directory*, and complete words. Based on Deißenböck and Pizka's observation, researchers have developed approaches to segment identifiers by splitting them into component terms.

Java developers often use English and the Camel Case convention to create identifiers. The Camel Case convention is the practice of creating identifiers by concatenating terms with capitalized first letter, giving identifiers a Camel-like looking with flats and humps, *e.g.*, *drawRectangle*. This practice is not very common in C programs where a mix of coding conventions are encountered. In particular, in C programs, terms are often concatenated into identifiers using an underscore as separator, *e.g.*, *draw_rectangle*.

These practices, Camel Case and underscore concatenations, lead to the development of a family of algorithms to split identifiers into component terms. These algorithms have in common the assumption that the Camel Case convention and–or an explicit separator are used systematically to create identifiers. Recently, a more complete strategy was implemented by the *Samurai* approach [EHPVS09], which uses a greedy algorithms and strings frequency tables to identify terms composing an identifier.

In the paper, we describe TIDIER (Term IDentifier RecognIzER), an approach that implements the algorithm summarized in our previous work [MGD$^+$10] and detailed in Section 2. TIDIER overcomes the limitations of any Camel Case splitter and of Samurai, in particular for C programs. Indeed, Camel Case splitters and Samurai have very good performance when applied on Java code because Java developers usually adhere strictly to the Camel Case convention. However, these approaches are not very effective on C identifiers [MGD$^+$10, EHPVS09].

Thus, this paper extends our previous work [MGD$^+$10] with the following contributions. First, we describe in more details our approach, TIDIER. Second, we perform an extensive validation of TIDIER on identifiers randomly extracted from a large set of C programs. Third, we compare TIDIER with a Camel Case splitter and Samurai. Finally, we provide evidence on the relevance of contextual information as well as specialized knowledge to the splitting process. The validation leads us to answer the following research questions:

Copyright © 2000 John Wiley & Sons, Ltd.

1. **RQ1:** *How does TIDIER compare with alternative approaches, Camel Case splitting and Samurai, when C identifiers must be split?*
2. **RQ2:** *How sensitive are the performances of TIDIER to the use of contextual information and specialized knowledge in different dictionaries?*
3. **RQ3:** *What percentage of identifiers containing word abbreviations is TIDIER able to map to dictionary words?*

To answer these research questions, we analyzed a set of 340 open-source programs: 337 programs from the GNU repository, two operating systems (the Linux Kernel release 2.6.31.6 and FreeBSD release 8.0.0), and the Apache Web server release 2.2.14. We extracted identifiers (including function, parameter, and structure names) and comments from these programs. We manually built an oracle of 1,026 identifiers randomly extracted from the 340 programs and not being plain English words, well-known acronyms, or terms. Using this oracle, we report evidence of the superiority of TIDIER over a Camel Case splitter and Samurai for C identifiers (**RQ1**). We show the usefulness to TIDIER of contextual information and specialized knowledge (**RQ2**). We also provide supporting evidence that TIDIER successfully expands abbreviations in about 48% of cases (**RQ3**).

As in our previous work [MGD$^{+}$10], 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[2] associate multiple meanings to words).

This paper is organized as follows. Section 2 describes TIDIER. Section 3 overviews two alternative approaches, a Camel Case splitter and Samurai [EHPVS09]. Section 4 describes the empirical study aimed at evaluating the proposed approach. The study results are reported and discussed in Section 5 while Section 6 provides a discussion of the threats to their validity. Section 7 relates this work to the existing literature. Finally, Section 8 concludes and outlines future work.

## 2. Proposed Approach

When writing source code, in particular when naming source-code identifiers, developers make use of concepts from high-level documentation and from the program domain. Also, they encode identifiers using implicit and explicit coding conventions and–or past experience. The goal of TIDIER is to split program identifiers using high-level and domain concepts by associating identifier terms to domain-specific words or to words belonging to some generic English dictionary.

First, TIDIER assumes that it is possible to model developers creating an identifier with a set of transformation rules on terms/words. For example, to create an identifier for a variable that stores a number of customers, the two words *number* and *customers* can be concatenated with

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

or without an underscore, *e.g.*, *customer_number* or *customernumber* or following the Camel Case convention, *e.g.*, *customerNumber*. Contractions of one or both words are also possible, leading to identifiers such as *customerNbr*, *nbr_customer*, *nbrOfCustomer*, or *cstmr_nbr*.

Second, TIDIER assumes that it is possible to define a distance between dictionary words and identifier terms to quantify how close are words, representing concepts, to such terms and, thus, to provide a measure of the likelihood that the terms refer to some words. Although there are several ways—none of which the best—to compute a distance between two terms and words, string-based distances have been used in the past for various purposes, such as code differencing [CCD09] or clone tracking [TCAD10], with good results. Therefore, TIDIER use the string-edit distance between terms and words as a proxy for the distance between the terms and the concepts they represent.

In a nutshell, TIDIER relies on a set of input dictionaries and a distance function to split (if necessary) simple and composed identifiers and associate the resulting terms with words in the dictionaries, even if the terms are truncated/abbreviated, *e.g.*, *objectPtr*, *cntr*, or *drawrect*. Dictionaries may include English words and–or technical words, *e.g.*, *microprocessor* and *database* (in the computer domain), or known acronyms, *e.g.*, *afaik* (in the Internet jargon). The distance function measures how close a given identifier term is to a dictionary word and, thus, how well the concepts associated to the dictionary words are conveyed by the identifier.

Developers of C programs sometime use word abbreviations to compose identifiers, which is likely an heritage of the past when certain operating systems and compilers limited the maximum length of identifiers. For example, a developer may use the term *dir* instead of the word *directory*, *ptr* or *pntr* instead of *pointer*, or *net* instead of *network*. TIDIER aims at segmenting identifiers into terms and recovering the original non-abbreviated words. Thus, TIDIER uses a thesaurus rather than English and–or domain dictionaries. A thesaurus entry, a word, in TIDIER is the original word followed by the list of abbreviated terms *i.e.*, word synonyms; if TIDIER finds the term *ptr* in an identifier, then it knows that this term is actually an abbreviation of *pointer*. In the following, wherever there is no risk of confusion, the two terms dictionary and thesaurus will be used interchangeably to indicate a list of words; each word possibly associated with list of abbreviations.

Some abbreviations are well-known and can thus be part of the thesaurus *e.g.*, *directory* and *dir*. Some other abbreviations may not appear in the thesaurus because they are too domain and–or developer specific. To cope with such abbreviations, TIDIER is the first approach that find the best splitting using a string-edit distance and a greedy search. If the edit distance between a term and a word is not zero, TIDIER tries to reduce the distance by transforming the word into some possible abbreviated forms, *e.g.*, by removing all vowels *pointer* is mapped into *pntr*. Then, TIDIER recomputes the edit distance and adds the abbreviated forms as possible synonym of the word if the distance between the abbreviated form and the term is smaller than the previous distance between the word and the term. A hill-climbing algorithm iterates over all words and all transformation rules to obtain the best split—*i.e.*, a zero distance—or until a termination criterion is reached.

Thus, the current implementation of TIDIER takes as input an identifier and a thesaurus and uses a simple string-edit distance to split, whenever possible, the identifier into a number of terms that have a small (or zero) distance with dictionary words. TIDIER is not able to deal with missing information or to generate abbreviations in all cases. If the identifiers use

Copyright © 2000 John Wiley & Sons, Ltd.

| | | | | | | |
|---|---|---|---|---|---|---|
| **8** | r | ∞ | 6 | 5 | 4 | 3 |
| **7** | e | ∞ | 5 | 4 | 3 | 4 |
| **6** | t | ∞ | 4 | 3 | 2 | 3 |
| **5** | n | ∞ | 3 | 2 | 5 | 6 |
| **4** | i | ∞ | 2 | 3 | 4 | 5 |
| **3** | o | ∞ | 1 | 2 | 3 | 4 |
| **2** | p | ∞ | 0 | 1 | 2 | 3 |
| **1** | | 0 | ∞ | ∞ | ∞ | ∞ |
| | | | p | n | t | r |
| | | **1** | **2** | **2** | **2** | **5** |

Figure 1. Single Word Edit Distance Example

terms belonging to a specific domain, whose words are not present in the thesaurus, TIDIER cannot split and associate these terms with words. Similarly, TIDIER cannot identify the words composing acronyms, *e.g.*, *afaik*, *cpu*, *ssl*, or *imho*, because it cannot associate a single letter from the acronym with the corresponding word: for any letter, there exist thousands of words with the same string-edit distance, *e.g.*, the *c* of *cpu* has the same distance with *central* and with any other word starting with *c*.

We now detail the main components of TIDIER.

## 2.1. String Edit Distance

The string-edit distance between two strings, also known as Levenshtein distance [Lev66], is the number of operations required to transform one string into another. The most common setting considers the following edit operations: character deletion, insertion, and substitution. Specifically, these settings assume that each insertion and each deletion increase the distance between the two strings by one, whereas a substitution (*i.e.*, a deletion followed by one insertion) increases it by two [CLR90]. An exact match is just a special case of substitution; it has a zero cost because $c(i, j) = 0$ as both character $s[i]$ and $w[j]$ are the same.

Let us assume that we must compute the edit distance between the strings *pointer* and *pntr*. Their edit distance is three, as the characters *o*, *i*, and *e* must be removed from *pointer* or, alternatively, added to *pntr*. The main problem in computing the string-edit distance is the algorithm efficiency. A naive implementation is typically exponential in the string length. A quadratic complexity implementation can be easily realized using dynamic programming and the algorithm is then often referred to as the Levenshtein algorithm. The Levenshtein algorithm computes the distance between a string $s$ of length $N$ and a string $w$ of length $M$ as follows.

First, a distance matrix $D$ of $(N + 1) \times (M + 1)$ cells is allocated; in our example, $8 \times 5$, *i.e.*, the lengths of *pointer* and *pntr* plus ones. The cells in the first column and first row are initialized to a very high value but for cell $(1, 1)$, which is initialized to zero. (This allocation and initialization strategy simplifies the algorithm implementation.) Matrix $D$ can be seen as

Copyright © 2000 John Wiley & Sons, Ltd.

**Columns**

| Rows | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **4** | r | ∞ | 2 | 3 | 2 | 1 | 3 | 3 | 4 | 3 |
| **3** | t | ∞ | 1 | 2 | 1 | 2 | 3 | 3 | 3 | 4 |
| **2** | p | ∞ | 0 | 1 | 2 | 3 | 2 | 3 | 4 | 3 |
| **1** | | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | | | p | n | t | r | c | n | t | r |
| | | | | | | | | | | |
| **5** | r | ∞ | 4 | 4 | 3 | 2 | 3 | 3 | 2 | 1 |
| **4** | t | ∞ | 3 | 3 | 2 | 3 | 3 | 2 | 1 | 2 |
| **3** | n | ∞ | 2 | 2 | 3 | 3 | 2 | 1 | 2 | 3 |
| **2** | c | ∞ | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 3 |
| **1** | | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | | | p | n | t | r | c | n | t | r |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Minimal Distance | ∞ | 2 | 3 | 2 | 1 | 2 | 3 | 1 | 1 |

Figure 2. Multiple Words Edit Distance Example

a Cartesian plane, and strings $s$ and $w$, *i.e.*, *pointer* and *pntr*, as places along the plane axes starting from the second cells, as shown in Figure 1.

The computation proceeds column by column starting from cell $(1, 1)$. The distance in cell $D(i, j)$ is computed as a function of the previously computed (or initialized) distances in cells $D(i-1, j)$, $D(i-1, j-1)$, and $D(i, j-1)$. At the end of the process, the cell $(N+1, M+1)$ contains $D(N+1, M+1)$, which is the minimum edit distance.

$$\begin{array}{rr l}
c(i,j) = & \begin{cases} 1 & \text{if s[i]} \neq \text{w[j]} \\ 0 & \text{if s[i]} = \text{w[j]} \end{cases} & \\
D(i,j) = & min[D(i-1,j) + c(i,j), & // \textit{ insertion} \\
 & D(i,j-1) + c(i,j), & // \textit{ deletion} \\
 & D(i-1,j-1) + 2 * c(i,j)] & // \textit{ substitution}
\end{array}$$

Unfortunately, the Levenshtein algorithm is not suitable to split identifiers because it only computes the distance between two given strings, not between sub-strings in a string (*i.e.*, identifier terms) and some other strings (*i.e.*, dictionary words).

In the early '80s, Ney proposed [Ney84] an adaptation to continuous speech recognition of the dynamic programming alignment algorithm, known as Dynamic Time Warping (DTW) [SC78], originally conceived for isolated word recognition. Ney's adaptation considers that a word can begin and end at any point in an utterance, similarly as a term can begin and end at any point in an identifier. It thus does not assume an a-priori knowledge of where a word is located in an utterance, *i.e.*, where a term begins or ends in an identifier. The details of Ney's algorithm are available elsewhere [Ney84]. TIDIER implements an extension of the Levenshtein algorithm based on Ney's adaptation. This extension requires a dictionary (or a thesaurus) of known words (referred to as *speech template* in [Ney84, SC78]).

Let us suppose that we have the identifier *pntrcntr* and that our dictionary contains only the two words *ptr* and *cntr*, abbreviations of *pointer* and *counter*, respectively. The algorithm initialized as described above for the Levenshtein algorithm, except that it creates one matrix for each word in the dictionary, as shown in Figure 2. The algorithm then proceeds by computing one column at a time, going from row 2, to row $N + 1$. Row 1 and Column 1 just contain initialization values used to simplify the DTW and, thus, the actual computation goes from cell $(2, 2)$ to cell $(N + 1, N + 1)$.

Once Column 2 is computed for all words in the dictionary as in the Levenshtein algorithm, a decision is taken on the minimum distance contained in cell $(2, 4)$ for *ptr* and $(2, 5)$ for *cntr*. This minimum distance is equal to 2, as shown in Figure 2, and the corresponding best term, *i.e.*, *ptr*, is then recorded. (The best term is not reported in Figure 2 due to space reason.) The minimum distance is then copied into the cell $(1, 3)$ of the matrices, which corresponds to assuming that the word with lower cost ends at Column 2.

At the beginning of Column 3 (*i.e.*, to calculate $(2, 3)$), the algorithm checks if it is less costly to move from one of the cells $(1, 2)$ and $(2, 2)$ or, instead, if it is cheaper to assume that a string was matched at column two (previous column) with the distance cost recorded in the minimum distance array (*i.e.*, two) and copied into $(1, 3)$. In the example, for both dictionary words, the algorithm decides to insert a character, *i.e.*, move to the next column (along the $x$ axis), as previous values are lower, *i.e.*, zero for *ptr* and one for *cntr*.

When the column of the character *c* of *pntrcntr* is computed (Column 6), the minimum distance recorded for dictionary terms at Column 5 is one, as *ptr* just needs one character insertion to match *pntr*. Thus, the computation propagates the minimum distance in Column 5 for *ptr*, *i.e.*, *ptr* matches *pntr* with distance one, and the algorithm detects that the word *ptr* ends at Column 5. Because the character *c* is matched in *cntr*, the distance of one is propagated to cell $(6, 2)$. The last part of the identifier *pntrcntr* matches *cntr*. Thus, when all columns are computed, the lowest distance is one. Distance matrices and the minimum distance array allow to compute the minimum string-edit distance between the terms in the identifier and the two words, and thus split the identifier.

The algorithm uses back-pointers matrices for improved performance, one for each dictionary word. For a given term, in each cell $(i, j)$, the back-pointer matrix records the decision taken and thus words, words matching distances, as well as the begin and end of each word is recovered.

### 2.2. Word Transformation Rules

However, some identifier terms may not be part of the thesaurus and must be generated from existing words and, possibly, added to the thesaurus. Let us consider the identifier *fileLen* and suppose that the thesaurus contains the words *length*, *file*, *lender*, and *ladder*, and no abbreviations. Clearly, the word *file* matches with a zero string-edit distance with 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 deleted. 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 be associated with the term *len.*

We define and use the following transformation rules in TIDIER:

- *Delete a random character:* one randomly-chosen character is deleted from the word, *e.g.*, *pointer* becomes *poiner*;
- *Delete a random vowel:* one randomly-chosen vowel from the word is deleted, *e.g.*, *number* becomes *numbr*;
- *Delete all vowels:* all the vowels in a word are deleted, *e.g.*, *pointer* becomes *pntr*;
- *Delete suffix:* the suffix of the word, such as *ing*, *tion*, *ed*, *ment*, *able*, is deleted, *e.g.*, *improvement* becomes *improve*;
- *Keep the first n characters only:* the word is transformed by keeping the first $n$ characters only, *e.g.*, *rectangle* becomes *rect*, with $n = 4$.

Constraints exist between transformation rules. For example, it is impossible to delete a random vowel once all vowels have been deleted; a suffix can be removed if and only if it is part of the word.

To choose the most suitable word to be transformed, TIDIER uses the following simple heuristic. It selects the closest words to the term to be matched, *i.e.*, smallest non-zero distances, and repeatedly transforms these words using randomly-chosen transformation rules. This process continues until a transformed word matches the term or the transformed words reach a length shorter than or equal to three characters. We choose three characters as lower limit because too many words would have the same abbreviation with two or less characters. If the transformed word matches the term, then this abbreviation is added in the thesaurus, else the algorithm tries to transform the next closest words to either find an abbreviation or reports a failure to match the term with any word/abbreviation.

### 2.3. Thesaurus of Words and Abbreviations

The thesaurus used by TIDIER plays an important role for the quality of its results. In the thesaurus, a word may be followed by a list of equivalent words or abbreviations. For example, the words *network* and *net* are considered equivalent and form a single row as well as the terms *pointer*, *pntr*, and *ptr*. Thus, if *pntr* is matched, TIDIER expands it into the dictionary word *pointer*.

One possibility to build such a thesaurus would be to merge different specific or generic dictionaries, such as those of spell checkers, *e.g.*, i-spell[3], which contains about 35,000 words, or of upper ontologies, *e.g.*, WordNet[4], which contains about 90,000 entries.

Yet, it would be desirable, if possible, to build smaller dictionaries, *e.g.*, dictionaries containing the most frequently-used English words only as well as specialized dictionaries

[3] `http://www.gnu.org/software/ispell/ispell.html`
[4] `http://wordnet.princeton.edu/wordnet/`

Copyright © 2000 John Wiley & Sons, Ltd.

containing acronyms and known abbreviations to reduce computation times. In the following, we use five different kinds of dictionaries.

1. *Small English dictionary (referred to as "English Dictionary"*: an English dictionary built from the 1,000 most frequent English words, the 250 most frequent technical words (from the Oxford Dictionary), and 275 most frequent business words (from the Oxford Dictionary) plus words from a glossary found on the Internet[5]. Overall, this dictionary includes 2,774 words.
2. *Small English dictionary, plus specialized knowledge*: this dictionary consists of the English Dictionary plus: (i) a set of 105 acronyms used in computer science (*e.g.*, *ansi*, *dom*, *inode*, *ssl*, *url*), (ii) a set of 164 abbreviations collected among the authors used when programming in C (*e.g.*, *bool* for *boolean*, *buff* for *buffer*, *wrd* for *word*), and (iii) a set of 492 C library functions (*e.g.*, *malloc*, *printf*, *waitpid*, *access*). This dictionary includes the union of the 2,774 English words plus 761 abbreviations and C functions, for a total of 3,535 distinct words.
3. *Complete English dictionary (referred to as "WordNet"*: a complete English dictionary extracted from the WordNet upper-ontology database and from the GNU i-spell spell-checker. This dictionary includes 175,225 words.
4. *Context-aware dictionaries*: similarly to Enslen *et al.* [EHPVS09], dictionaries containing function-level, source code file-level, and program-level identifiers. We built these dictionaries using words appearing in the context where the identifiers are located.
5. *Application dictionary, plus specialized knowledge*: a dictionary based on the program-level dictionary—described in the previous step—augmented with domain knowledge (abbreviations, acronyms, and C library functions).

The abbreviations used to describe specialized knowledge were collected with no prior knowledge about the identifiers to be split. The rationale of including abbreviations is to identify terms not contained in the English Dictionary but that are likely to be contained in identifiers and that could not be expanded into English words because their distance from the words that they represent is too large. For example, the identifier *ipconfig* contains the term *ip*, which means "internet protocol". It would be impossible for any algorithm to guess that *i* stands for *internet* and *p* for *protocol.* Widely-used abbreviations are introduced to make the search faster as it would be useless and time consuming to generate well-known abbreviations. C library terms are introduced because, often, they correspond to jargon or domain-specific words and C program identifiers contain these terms. For example, functions wrapping known C functions often contain terms such as *printf*, *socket*, *flush*, and so on, as in the Linux identifiers *threads__fprintf*, *seq_printf*, or, in the Apache Web server, *snprintf_flush* or *apr_socket_create.*

The context-aware dictionaries are built by tokenizing source code, extracting identifiers and comment terms, and saving them into specialized context-aware dictionaries at the level of functions, files, or programs. These lists of terms need to be pruned of strings not corresponding

[5] `http://www.matisse.net/files/glossary.html`

Copyright © 2000 John Wiley & Sons, Ltd.

to English words or technical terms before being considered as usable dictionaries; in TIDIER, the filtering is done by string comparison with the WordNet dictionary.

TIDIER dictionaries must be carefully validated as its results depend on them. Building and validating dictionaries is a non-trivial activity. We use two ways to validate a dictionary. Manual validation for small dictionaries or highly-specific dictionaries, such as the abbreviations, and automatic filtering using a trusted reference dictionary, among others WordNet, for large dictionaries.

Typically, we create a dictionary for expanding identifiers as follows:

- First, we create a dictionary containing words from the English language (assuming that identifiers are in English) using already-available dictionaries, such as GNU i-spell or WordNet.
- Second, we build context-aware dictionaries by filtering WordNet/i-spell words that appear in a given source-code context. We use source code tokenization and pattern matching to perform the filtering automatically.
- Finally, we complement the previously-built dictionaries with domain-specific words (not contained in the original dictionaries) and acronyms (together with their expansions), which is the the most critical task.

Overall, the four authors spent about one day to produce TIDIER dictionaries, populated with abbreviations and acronyms typical for Unix utilities (i.e., the domain of our empirical study). In addition, we spent a couple of days to document the C library functions through tedious manual verification.

### 2.4. Putting It All Together

We now describe a typical run of TIDIER. First, wherever possible, identifiers are split using explicit separators, namely special characters, *e.g.*, "_", ".", "$", "->", and the Camel Case convention. (We describe a Camel Case splitter in Section 3.1.)

Then, TIDIER applies transformations and computes the distance between the identifier terms and the thesaurus words by using a hill climbing search. For a given identifier and a given dictionary, the string-edit distance assigns a distance to each thesaurus word as well as the positions where it begins and ends in the identifier. The edit distance is the fitness function guiding the hill-climbing search as follows:

1. Based on the thesaurus, TIDIER (i) splits the identifier using the edit distance, (ii) computes the global minimum distance between the input identifier and all words in the thesaurus, (iii) associates a fitness value based on the distance computed in step (ii) to each thesaurus word. If the minimum global distance in step (ii) is zero, the process terminates successfully; else
2. From the thesaurus words with non-zero distance obtained at Step 1, TIDIER randomly selects one word having a minimum distance and:
   (a) TIDIER randomly selects one transformation not violating transformation constraints, applies it to the word, and adds the transformed word to a temporary thesaurus;

Copyright © 2000 John Wiley & Sons, Ltd.
*Prepared using* smrauth.cls

(b) TIDIER splits the identifier using the temporary thesaurus and computes a new minimum global distance. If the added transformed word reduces the previous global distance, then TIDIER adds it to the current thesaurus and go to Step (a); else

(c) If there are still applicable transformations, and the string produced in Step (a) is longer then three characters, TIDIER goes to Step (a);

3. If the global distance is non-zero and the iteration limit was not reached, then, TIDIER goes back to Step 1, otherwise it terminates with a failure.

The above steps describe a hill-climbing algorithm, in which a transformed term is added to the thesaurus if and only if it reduces the global distance. Briefly, a hill-climbing algorithm [MF04] searches for a (near) optimal solution of a problem by moving from the current solution to a randomly chosen, nearby solution and accepts this solution only if it improves the solution global fitness. The algorithm terminates when there is no move to nearby solutions improving the fitness. Differently from traditional hill-climbing algorithms, in Steps 1 and 2, TIDIER attempts to explore as many neighboring solutions as possible by performing word transformations. Different neighbors are explored depending on the order of the transformations.

Currently, the implementation of TIDIER uses a naive strategy to select a transformation. However, in our experience, even such a strategy performs well with small-to-medium size dictionaries (up to 5,000 words). For dictionaries larger than 20,000 words, the computation time to obtain meaningful results can become excessive. For example, with a dictionary of about 100,000 words and an upper computation limit of 20,000 attempts to improve distance, the computation can take up to 30 minutes or one hour depending on the input identifier. Therefore, we suggest restarting the search process several times with an upper bound for the computations ranging from 5,000 to 10,000.

### 2.5. Discussion on TIDIER

Using techniques inspired from dynamic programming and string-edit distance is not the only way of splitting identifiers into words. Clearly, when developers used explicit separators, there is no need for complex splitting algorithms. However:

- In some situations, the Camel Case convention or other explicit separators are not used, thus automatic approaches must be used. A possible alternative is the approach by Enslen *et al.* [EHPVS09], described in Section 3.2.
- The extended string-edit distance used in TIDIER provides a distance between an identifier and a set of words in a dictionary, even if there is no perfect match between terms in the identifier and dictionary words; for example, when identifiers are composed of abbreviations, *e.g.*, *getPntr*, *filelen*, or *pntrctr*. TIDIER accepts a match by identifying the dictionary words closest to the identifier terms.
- The DTW algorithm performs an alignment when matching words from the dictionary with terms in an identifier, thus it works even when the word to be matched is preceded

Copyright © 2000 John Wiley & Sons, Ltd.

or followed by other characters, *e.g.*, *xpntr*. Thus, it is better than, for example, applying the Levenshtein edit distance only.
- The string-edit distance algorithm assigns a distance to matched terms. Thus, in the above *fileLen* example, we would discover that *file* matches the first four characters with a zero distance and that *length* matches the five to seven characters (with a distance of three).
- The dictionary can be sorted so that the approach favors the matching of the longest words rather than matching the multiple words composing the longest one. Thus, the identifier *copyright* would be matched to the word *copyright* rather than to the words *copy* and *right*, which also belong to the dictionary.

Although a distance-based identifier splitting approach is promising, it does consider, *per se*, neither semantics nor contextual information. For example, with *fileLen*, *length* should be preferred over *lender*. However, the string-edit distance cannot be used to choose between *lender* or *length*.

Also, it is not possible to disambiguate complex identifiers that actually have an optimal non-zero distance splitting, because the algorithm always favors zero-distance splitting. For example, *imagEdges* contains the words *image* and *edges*. However, *image* and *edges* match the identifier with a distance of one because character $E$ is shared by both terms in the identifier. Clearly, in this example, developers would use syntax and semantics as well as contextual and specialized knowledge: even if *imag* is not an English word, they would correctly split *imagEdges* into *image* and *edges*.

Finally, the string-edit distance used by TIDIER has a cubic complexity in the number of characters in the identifier (say $M$), words in the dictionary (say $T$), and maximum number of characters composing dictionary words (say $N$). For each word in the dictionary, we must compute as many distances as there are cells to fill the distance matrix, with a complexity of $\mathcal{O}(M \times N)$. Because there are $T$ dictionary words, the overall complexity is $\mathcal{O}(T \times M \times N)$. A remarkable increase of performance can be achieved by saving the first edit-distance computation and, during the hill-climbing search, recomputing only cells where the distance improves.

## 3. Alternative Approaches

This section describes two state-of-the-practice and state-of-the-art approaches to split identifiers into terms. These approaches are the simple Camel Case splitter and Samurai by Enslen *et al.* [EHPVS09].

### 3.1. Camel Case Splitter

As a baseline for comparison, we implemented the state-of-the-practice identifier splitting algorithm: the Camel Case splitter. It splits identifiers according to the following rules:

**RuleA:** Identifiers are split by replacing underscore (*i.e.*, "_"), structure and pointer access (*i.e.*, "." and "->"), and special symbols (*e.g.*, $) with the space character. A space is

Copyright © 2000 John Wiley & Sons, Ltd.

inserted before and after each sequence of digits. For example, *counter_pointer4users* is split into *counter*, *pointer*, *4*, and *users* while *rmd128_update* is split into *rmd*, *128*, and *update*.

**RuleB:** Identifiers are split where terms are separated using the Camel Case convention, *i.e.*, the algorithm splits sequences of characters when there is a sequence of lower-case characters followed by one or more upper-case characters. For example, *counterPointer* is split into *counter* and *Pointer* while *getID* is split into *get* and *ID*.

**RuleC:** When two or more upper case characters are followed by one or more lower case characters, the identifier is split at the last-but-one upper-case character. For example, *USRPntr* is split into *USR* and *Pntr*.

**Default:** Identifiers composed of multiple terms that are not separated by any of the above separators are left unaltered. For example, *counterpointer* remains as it is.

Based on these rules, identifiers such as *FFEINFO_kindtypereal3*, *apzArgs*, or *TxRingPtr* are split into *FFEINFO kindtypereal*, *apz Args*, and *Tx Ring Ptr*, respectively. The Camel Case splitter cannot split *FFEINFO* or *kindtypereal* into terms, *i.e.*, the acronym *FFE* followed by *INFO* and the terms *kind*, *type*, and *real*. In our implementation of the Camel Case splitter, we do not model cases in which developers assigned a specific meaning to some characters, *e.g.*, the digits 2 and 4 could stand for the terms *to* and *for*, respectively, as in the identifiers *peer2peer* and *buffer4heap*. Furthermore, digits sequences and single character are pruned out.

### 3.2. Samurai Splitter

We also compare TIDIER with the state-of-the-art approach Samurai [EHPVS09]. Samurai is an automatic approach to split identifiers into sequences of terms by mining terms frequencies in a large source code base. It relies on two assumptions.

1. A substring composing an identifier is also likely to be used in other part of the program or in other programs alone or as part of other identifiers.
2. Given two possible splits of a given identifier, the split that most likely represents the developer's intent partitions the identifier into terms occurring more often in the program. Thus, term frequency is used to determine the most-likely splitting of identifiers.

Samurai also exploits identifier context. It mines term frequency in the source code and builds two term-frequency tables: a program-specific and a global-frequency table. The first table is built by mining terms in the program under analysis. The second table is made by mining the set of terms in a large corpus of programs.

Samurai ranks alternative splits of a source code identifier using a scoring function based on the program-specific and global frequency tables. This scoring function is at the heart of Samurai. It returns a score for any term based on the two frequency tables representative of the program-specific and global term frequencies. Given a term $t$ appearing in the program $p$, its score is computed as follows:

Copyright © 2000 John Wiley & Sons, Ltd.

$$Score(t, p) = Freq(t, p) + \frac{globalFreq(t)}{log_{10}(AllStrsFreq(p))} \tag{1}$$

where:

- $p$ is the program under analysis;
- $Freq(t, p)$ is the frequency of term $t$ in the program $p$;
- $globalFreq(t)$ is the frequency of term $t$ in a given set of programs; and
- $AllStrsFreq(p)$ is the cumulative frequency of all terms contained in the program $p$.

Using this scoring function, Samurai applies two algorithms, the *mixedCaseSplit* and the *sameCaseSplit* algorithm. It starts by executing the *mixedCaseSplit* algorithm, which acts in a way similar to the Camel Case splitter but also uses the frequency tables. Given an identifier, first, Samurai applies **RuleA** and **RuleB** from the Camel Case splitter: all special characters are replaced with the space character. Samurai also inserts a space character before and after each digit sequence. Then, Samurai applies an extension of **RuleC** to deal with multiple possible splits.

Let us consider the identifier *USRpntr*. **RuleC** would wrongly split it into *US* and *Rpntr*. Therefore, Samurai creates two possible splits: *US Rpntr* and *USR pntr*. Each possible term on the right side of the splitting point is then assigned a score based on Equation 1 and the highest score is preferred. The frequency of *Rpntr* would be much lower than that of *pntr*, consequently the most-likely split is obtained by splitting *USRpntr* into *USR* and *pntr*.

Following this first algorithm, Samurai applies the *sameCaseSplit* algorithm to find the split(s) that maximize(s) the score when splitting a same-case identifier, such as *kindtypereal* or *FFEINFO*. The terms in which the identifier is split can only contain lower-case characters, upper-case characters, or a single upper-case character followed by same-case characters.

The starting point of this algorithm is the first position in the identifier. The algorithm considers each possible split points in the identifier. Each split point would divide the identifier into a left-side and a right-side term. Then, the algorithm assigns a score for each possible left and right term and the split is performed where the split achieves the highest score. (Samurai uses a predefined lists[6] of common prefixes (*e.g.*, *demi*, *ex*, or *maxi*) and suffixes (*e.g.*, *al*, *ar*, *centric*, *ly*, *oic*) and the split point is discarded if a term is classified as a common prefix or suffix.)

Let us consider for example the identifier *kindtypereal* and assume that the first split is *kind* and *typereal*. Because neither *kind* nor *typereal* are common prefix/suffix, this split is kept. Now, let us further assume that the frequency of *kind* is higher than that of *kindtypereal* (*i.e.*, of the original identifier) and that the frequency of *typereal* is lower than that of *kindtypereal*. Then, the algorithm keeps *kind* and attempts to split *typereal* as its frequency is lower than that of the original identifier. When it will split *typereal* into *type* and *real*, the score of *type* and *real* will be higher than the score of the original identifier *kindtypereal* and of *typereal* and, thus, *typereal* will be split into *type* and *real*. Because the terms *kind*, *type*, and *real* have

[6] `http://www.cis.udel.edu/~enslen/Site/Samurai_files/`

Copyright © 2000 John Wiley & Sons, Ltd.

frequencies higher than that of *kindtypereal*, the obtained split corresponds to the expected result.

The main weakness of Samurai is its reliance on frequency tables. These tables could lead to different splits for the same identifier depending on the tables. Tables built from different programs may lead to different results. Also, if an identifier contains terms with frequencies higher than the frequency of the identifier itself, Samurai may split it into several terms not necessarily reflecting the most obvious split.

## 4. Empirical Evaluation

The *goal* of this study is to analyze the TIDIER identifier splitting approach with the *purpose* of evaluating its ability to adequately identify dictionary words composing identifiers, even in presence of abbreviations and–or acronyms. The *quality focus* is the precision and recall of the approach when identifying words composing identifiers with respect to a manually-built oracle and to alternative identifier-splitting approaches. The *perspective* is that of researchers, who want to understand if the approach for identifier splitting 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 terms or in general to meaningful words, *e.g.*, words belonging to a dictionary.

The *context* consists of a set of 1,026 composed identifiers randomly sampled from the source code of 337 GNU[7] projects, the Linux Kernel[8] 2.6.31.6, FreeBSD[9] 8.0.0, and the Apache Web server[10] 2.2.14. The GNU project was launched in 1984 with the ultimate goal to provide a free, open-source operating system and environment. GNU projects include well-known tools, such as the GCC compiler, parser generators, shells, editors, libraries, and textual utilities, just to name a few. Most code of the GNU project is written in C, with a few C++ program (*e.g.*, *groff*). Linux is the well-known operating system widely adopted on servers and, in recent years, used as a desktop alternative to proprietary operating systems. The Linux Kernel is entirely written in C with additional utilities written mostly in scripting languages, such as Bash or TCL/TK. FreeBSD is another freely available operating systems; as the name suggest it derives from the BSD branch of the Unix tree. The Apache Web server is a free and open-source Web server; it is adopted by public and private organizations for its robustness, speed, and security as well as its large community of developers. It is entirely developed in C. The main characteristics of these programs are listed in Table I.

### 4.1. Building the Dictionaries

TIDIER aims at splitting identifiers by trying to match their terms with words contained in a thesaurus. We use the different kinds of dictionaries introduced in Section 2.3. Specifically,

[7] `http://www.gnu.org/`
[8] `http://www.kernel.org/`
[9] `http://www.freebsd.org/`
[10] `http://www.apache.org/`

Copyright © 2000 John Wiley & Sons, Ltd.

Table I. Main characteristics of the 340 projects for the sampled identifiers

| GNU Projects (337 Projects) | | | | |
|---|---|---|---|---|
| | C | C++ | .h | Java |
| Files | 57,268 | 13,445 | 39,257 | 14,811 |
| Size (KLOCs) | 25,442 | 2,846 | 6,062 | 3,414 |
| Terms | 26,824 | – | 17,563 | – |
| Identifiers | 1,154,280 | – | 619,652 | – |
| Oracle Identifiers | 927 | – | 26 | – |
| Linux Kernel | | | | |
| | C | C++ | .h | Java |
| Files | 12,581 | – | 11,166 | – |
| Size (KLOCs) | 8,474 | – | 1,994 | – |
| Terms | 19,512 | – | 13,006 | – |
| Identifiers | 845,335 | – | 352,850 | – |
| Oracle Identifiers | 73 | – | 4 | – |
| FreeBSD | | | | |
| | C | C++ | .h | Java |
| Files | 13,726 | 128 | 7,846 | 15 |
| Size (KLOCs) | 1,800 | 128 | 8,016 | 4 |
| Terms | 21,357 | – | 12,496 | – |
| Identifiers | 634,902 | – | 278,659 | – |
| Oracle Identifiers | 20 | – | 0 | – |
| Apache Web Server | | | | |
| | C | C++ | .h | Java |
| Files | 559 | – | 254 | – |
| Size (KLOCs) | 293 | – | 44 | – |
| Terms | 6,446 | – | 3,550 | – |
| Identifiers | 33,062 | – | 11,549 | – |
| Oracle Identifiers | 11 | – | 0 | – |

Table II. Descriptive statistics of the contextual dictionaries

| Context | Min | 1Q | Median | 3Q | Max | Avg | $\sigma$ |
|---|---|---|---|---|---|---|---|
| Application | 29 | 900 | 1,797 | 3,028 | 22,190 | 2,320 | 2,374 |
| File | 1 | 40 | 79 | 175 | 4,088 | 131 | 148 |
| Function | 1 | 3 | 6 | 21 | 1,625 | 16 | 29 |

Table II reports descriptive statistics of the context-aware dictionaries, built from all programs from which identifiers were sampled.

## 4.2. Research Questions

The study reported in this section addresses the following research questions:

1. **RQ1:** *How does TIDIER compare with alternative approaches, Camel Case splitting and Samurai, when C identifiers must be split?* This research question analyzes the

performance of TIDIER and compares it with alternative approaches, a Camel Case splitter and an implementation of Samurai.

2. **RQ2:** *How sensitive are the performances of TIDIER to the use of contextual information and specialized knowledge in different dictionaries?* This research question analyzes the performances of TIDIER in function of different dictionaries.
3. **RQ3:** *What percentage of identifiers containing word abbreviations is TIDIER able to map to dictionary words?* This research question evaluates the ability of TIDIER to map identifier terms with dictionary words when these terms represent abbreviations of dictionary words.

### 4.3. Variable Selection and Study Design

The main independent variable of our study is the kind of splitting algorithm being used. There are three different values for this factor:

1. Camel Case splitter;
2. Samurai approach;
3. TIDIER approach.

The second independent variable is the used dictionary (or set of dictionaries) among those defined in Section 2.3. Thus, we have a number of possible treatments equals to the number of different dictionaries plus two, *i.e.*, the two alternative approaches: Camel Case splitter and Samurai.

The first dependent variable considered in our study is the *correctness* of the splitting/mapping to dictionary words produced by the identifier-splitting approach with respect to the oracle. As a first, coarse-grain measure, we use a Boolean variable meaning that the split is correct (true) or not (false). Let us define the correct splitting of the identifier *cntrPtr* as *counter* and *pointer*; if the studied approach produces exactly the expected splits, then the correctness is true, else it is false, *e.g.*, *counter* and *ptr*. The weakness of this correctness measure is that it only provides a Boolean evaluation of the splitting. If the split is *almost* correct, *i.e.*, most of the terms are correctly identified, then correctness would still be false.

To overcome the limitation of the correctness measure and provide a more insightful evaluation, we use the precision and recall measures. Given an identifier $s_i$ to be split, $o_i = \{oracle_{i,1}, \ldots oracle_{i,m}\}$ the splitting in the manually-produced oracle, and $t_i = \{term_{i,1}, \ldots term_{i,n}\}$ the set of terms obtained by an approach, we define precision and recall as follows:

$$precision_i = \frac{|t_i \cap o_i|}{|t_i|}, \quad recall_i = \frac{|t_i \cap o_i|}{|o_i|}$$

To provide an aggregated, overall measure of precision and recall, we use the F-Measure, which is the harmonic mean of precision and recall:

$$\mathrm{F-Measure} = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

### 4.4. Building the Oracle

To evaluate the performances of the TIDIER approach (*i.e.*, precision, recall, and F-measure) and to compare it with alternative ones, we must have an oracle, *i.e.*, for each identifier, a list of terms obtained after splitting it and, wherever needed, after expanding contracted words. For example, a possible oracle for *counterPntr* would be *counter pointer*, obtained by splitting the identifier after the seventh character and after expanding the abbreviation *Pntr* into *pointer*. Ideally, a perfect oracle could have been produced by the system developers; however, because this was infeasible to achieve for the hundreds of GNU projects from which we sampled the identifiers, the oracle has been produced by two of the authors as follows: (i) each author produced independently a splitting of each sampled identifier and expanded abbreviations; (ii) for all cases where there was a different splitting/expansion, a discussion was held and a consensus was reached. If a consensus was not possible and contrasting evidence was found, the identifier was removed from the oracle. The manual analysis was performed relying on various sources of information of the projects, ranging from source-code comments to user manuals. While sampling identifiers to create our oracle, identifiers corresponding to a single domain term or English word were discarded as they are not meaningful to answer the research questions and another (compound) identifier was added to the sample. Consequently, out of an initial set of 1,061 identifiers, we built an oracle of 1,026 identifiers. Eliminated identifiers were, for example, acronyms of two letters, such as "BC" for Binary Calculator, or non-English words, such as "sollte", the German word for "ought" (see file style.c of the GNU application diction version 1.1).

### 4.5. Analysis Method

**RQ1** and **RQ2** concern the comparison of the correctness, precision, recall, and F-Measure of the different approaches and of variations of TIDIER when using different dictionaries. Thus, the analysis methods are the same for both research questions and their results are presented together in the next section.

We test the differences among different approaches using the Fisher's exact test because correctness is a categorical measure. We test the following null hypothesis $H_0$*: the proportion of correct splits, $p_1$ and $p_2$, between two approaches do not significantly change.*

To quantify the effect size of the difference between any two approaches, we also compute the odds ratio ($OR$) [She07] 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 the first approach, and of the odds $q$ of the event to occur in the other sample, *i.e.*, the percentage of identifiers correctly split by the second approach: $OR = \frac{p/(1-p)}{q/(1-q)}$. $OR = 1$ indicates that the event is equally likely in both samples. $OR > 1$ indicates that the event is more likely with the first approach while an $OR < 1$ indicates the opposite.

Precision, recall, and F-Measure are compared using a non-parametric test for pair-wise median comparison, specifically the Wilcoxon paired test. We use a paired test as our samples are dependent, as we compute, for each identifier, the precision, recall and F-Measure for the different approaches. The Wilcoxon test reports whether the median difference between two approaches is significantly different from zero: $H_0 : \mu_d = 0$, where $\mu_d$ is the median of the differences.

We quantify the effect size of the difference using the Cohen $d$ effect size for dependent variables, defined as the difference between the means ($M_1$ and $M_2$), divided by the standard deviation of the (paired) differences between samples ($\sigma_D$):

$$d = \frac{M_1 - M_2}{\sigma_D}$$

The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$ [Coh88]. We choose the Cohen $d$ effect size because it is appropriate for our variables (in ratio scale) and because the different levels (small, medium, and large) are easy to interpret.

As both the Fisher's exact test and the Wilcoxon paired test are executed multiple times to compare the various approaches and dictionaries, significant $p$-values must be corrected. We used the Holm correction [Hol79], which is similar to the Bonferroni correction, but less stringent. It works as follows: (i) the $p$-values obtained from multiple tests are ranked from the smallest to the largest, (ii) the first $p$-value is multiplied by the number of tests performed ($n$), and is deemed to be significant if it is less than 0.05, and (iii) the second $p$-value is multiplied by $n - 1$, and so on.

For **RQ3**, we identify a set of abbreviations used in the sampled identifiers and compute the percentage of these abbreviations that are correctly mapped to dictionary words by TIDIER. We identify the set of likely abbreviations in our sample as follows:

1. For each identifier, *e.g.*, *counterPtr*, we consider the split performed using the Camel Case splitter, *i.e.*, *counter ptr*, and the oracle, *i.e.*, *counter pointer*;
2. Then, we compare each term in the split with the term appearing in the same position in the oracle, *e.g.*, *counter* is compared with *counter* and *ptr* with *pointer*;
3. For all cases where (i) the term in the splitting does not match with the one in the oracle, (ii) both terms start with the same letter, (iii) the term in the splitting does not appear in the English dictionary of 2,774 words, and (iv) the term in the oracle appears in the English dictionary, we consider the term in the splitting as an abbreviation of the term in the oracle: *ptr* is an abbreviation of *pointer*.

The set of 73 abbreviations obtained with the above process has been manually validated to remove false positive. Then, we apply each approach, considering the English dictionary with domain knowledge, and count the percentage of abbreviations correctly mapped to dictionary words. We also compute the set of abbreviations that are not correctly mapped, but with a distance of one from the oracle, *i.e.*, the mapping failed for a single character only. Thus, we identify and discuss cases where the approach almost found the correct solution, even though it failed to correctly split the identifier.

Table III. Comparison among approaches: results of Fisher's exact test and odds ratios

| Approach 1 | Approach 2 | $p$-values | $OR$s |
|---|---|---|---|
| Camel Case | Samurai | 0.63 | 0.95 |
| English dictionary | Camel Case | **0.01** | 0.73 |
| English dictionary | Samurai | **0.01** | 0.69 |
| English dictionary | WordNet | 1.00 | 0.95 |
| English dictionary + domain kn. | Camel Case | <**0.001** | 1.53 |
| English dictionary + domain kn. | Samurai | <**0.001** | 1.46 |
| English dictionary + domain kn. | English dictionary | <**0.001** | 2.13 |
| Application | Camel Case | 1.00 | 1.06 |
| Application | Samurai | 1.00 | 1.01 |
| Application | English dictionary + domain kn. | <**0.001** | 0.69 |
| Application | File | <**0.001** | 2.98 |
| Application | Function | <**0.001** | 7.86 |
| File | Function | <**0.001** | 2.63 |
| Application + Domain kn. | Application | <**0.001** | 2.56 |
| Application + Domain kn. | English dictionary | <**0.001** | 3.80 |
| Application + Domain kn. | English dictionary + domain kn. | <**0.001** | 1.80 |
| Application + Domain kn. | Camel Case | <**0.001** | 2.76 |
| Application + Domain kn. | Samurai | <**0.001** | 2.62 |

## 5. Experimental Results and Discussions

We now present and discuss the results of our study to answer the research questions formulated in Section 4.2. Raw data of our study are available on-line[11] for replication purposes.

### 5.1. RQ1 and RQ2

First, we evaluate the correctness of TIDIER when using different dictionaries and compare it with that of the two alternative approaches, *i.e.*, the Camel Case splitter and Samurai. We report the percentages of correctly split/mapped identifiers in Figure 3.

The two bars at the bottom of the figure show the performances of the Camel Case splitter and Samurai, respectively, while the other bars show the performances of TIDIER using different dictionaries.

Table III reports results of the Fisher's exact test (with corrected $p$-values, significant $p$-values are shown in bold face) when performing a pair-wise comparison among approaches of the percentages of correctly split identifiers. The table also reports the $OR$s. $OR$s greater than one indicate results in favor of Approach 1 and vice versa.

Figure 3 and Table III show that:

[11] http://web.soccerlab.polymtl.ca/ser-repos/public/TIDIER-rawdata.tgz

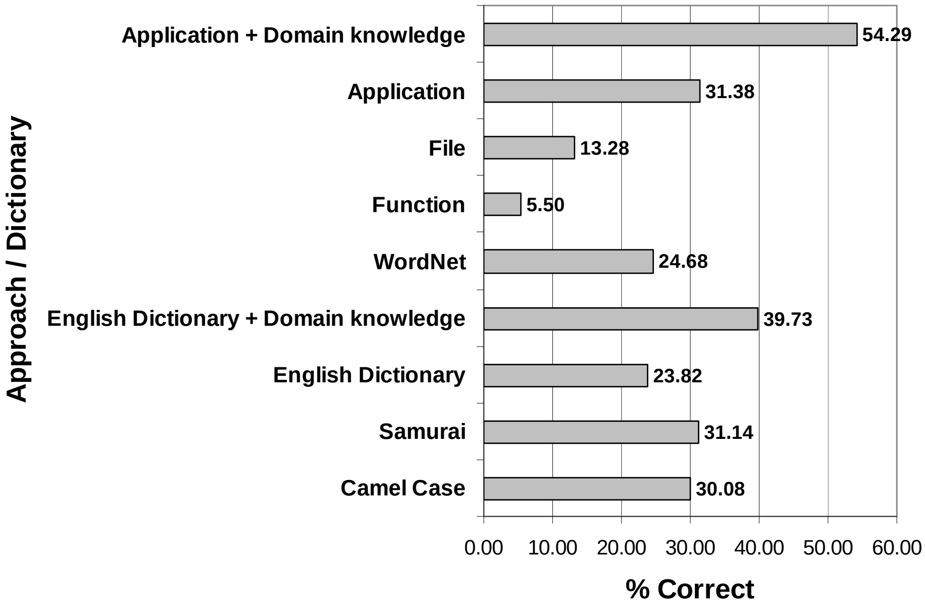

Figure 3. Percentages of correctly-split identifier

- In the extracted sample, Samurai performs nearly as well as the Camel Case splitter and there are no statistically significant differences among them.
- When using only the simple English dictionary, TIDIER performs worse than the Camel Case splitter and Samurai. The percentage of correctly-split identifiers is only 23.82%, while the Camel Case splitter exhibits a performance of 30.08% and Samurai of 31.14%. The $OR$ for TIDIER is 0.73 and 0.69 with respect to the two alternatives.
- When using a larger dictionary, *i.e.*, the WordNet dictionary, TIDIER does not perform significantly better (nor worse) than when using the simple English dictionary.
- When domain knowledge is added to the English dictionary, TIDIER significantly outperforms the alternative approaches. The percentage of correctly-split identifiers is nearly 40% with $OR$s of about 1.5 in favor of TIDIER wrt. the Camel Case splitter and Samurai.
- When using a contextual, program-level dictionary, TIDIER performs slightly (but not significantly) better (31.38%) than the alternative approaches but worse than when using the English dictionary with domain knowledge. Contextual dictionaries at file or function levels do not seem particularly useful because of their limited size and, thus, the number of terms that they capture.

Copyright © 2000 John Wiley & Sons, Ltd.
*Prepared using* smrauth.cls

Table IV. Descriptive statistics of F-Measure

| Method | Dictionary | 1Q | Median | 3Q | Mean | $\sigma$ |
|---|---|---|---|---|---|---|
| CamelCase | | 0.00 | 0.40 | 1.00 | 0.44 | 0.43 |
| Samurai | | 0.00 | 0.50 | 1.00 | 0.49 | 0.42 |
| TIDIER | English dictionary | 0.00 | 0.29 | 0.67 | 0.38 | 0.41 |
| | English dict. + domain kn. | 0.29 | 0.67 | 1.00 | 0.60 | 0.39 |
| | WordNet | 0.00 | 0.40 | 0.80 | 0.43 | 0.40 |
| | Function | 0.00 | 0.00 | 0.00 | 0.13 | 0.27 |
| | File | 0.00 | 0.00 | 0.57 | 0.30 | 0.37 |
| | Application | 0.00 | 0.50 | 1.00 | 0.52 | 0.40 |
| | Application + domain kn. | 0.50 | 1.00 | 1.00 | 0.72 | 0.36 |

Table V. Comparison among approaches: results of Wilcoxon paired test and Cohen *d* effect size

| Approach 1 | Approach 2 | *p*-value | *OR*s |
|---|---|---|---|
| Camel Case | Samurai | <**0.001** | −0.15 |
| English dictionary | Camel Case | <**0.001** | −0.12 |
| English dictionary | Samurai | <**0.001** | −0.19 |
| English dictionary | WordNet | <**0.001** | −0.11 |
| English dictionary + domain kn. | Camel Case | <**0.001** | 0.29 |
| English dictionary + domain kn. | Samurai | <**0.001** | 0.22 |
| English dictionary + domain kn. | English dictionary | <**0.001** | 0.61 |
| Application | Camel Case | <**0.001** | 0.18 |
| Application | Samurai | **0.01** | 0.10 |
| Application | English dictionary + domain kn. | <**0.001** | −0.16 |
| Application | File | <**0.001** | 0.46 |
| Application | Function | <**0.001** | 0.85 |
| File | Function | <**0.001** | 0.54 |
| Application + Domain kn. | Application | <**0.001** | 0.52 |
| Application + Domain kn. | English dictionary | <**0.001** | 0.81 |
| Application + Domain kn. | English dictionary + domain kn. | <**0.001** | 0.38 |
| Application + Domain kn. | Camel Case | <**0.001** | 0.58 |
| Application + Domain kn. | Samurai | <**0.001** | −0.51 |

- When adding domain knowledge to the program-level dictionary, TIDIER performs best with 54.29% of correct splits; percentage significantly higher than those of the alternative approaches and than that when using the English dictionary. *OR*s are 2.76 and 2.62 times in favor of TIDIER wrt. the Camel Case splitter and Samurai, respectively, and 1.80 wrt. using the English dictionary with domain knowledge.

Table IV shows the descriptive statistics (first quartile, median, third quartile, mean, and standard deviation) of the F-Measure computed as explained in Section 4.3 to evaluate the

Table VI. Summary of the performances of the various approaches/dictionaries. A higher number of "*" indicates better performances

| Approach | Performances |
|---|---|
| Camel Case | ** |
| Samurai | *** |
| TIDIER - English dictionary | * |
| TIDIER - English dictionary + domain knowledge | **** |
| TIDIER - Contextual (file) dictionary | * |
| TIDIER - Contextual (application) dictionary | *** |
| TIDIER - Contextual (application) dictionary + domain knowledge | ***** |

capability of the approaches to correctly and completely identify terms part of the identifiers. We do not show results of precision and recall separately (detailed tables are reported in Appendix 8) because they are consistent with the F-Measure, *i.e.*, there are no cases for which an approach exhibits a high precision and a low recall or vice versa.

Table V reports corrected results of the paired Wilcoxon test and the Cohen $d$ effect size (positive values of $d$ are in favor of Approach 1, negative values are in favor of Approach 2). Overall, these results are consistent with those obtained when measuring correctness. They show that:

- TIDIER, with the English dictionary, performs significantly worse than the other approaches with a very small effect size, $d < 0.2$.
- When using the English dictionary with domain knowledge, TIDIER performs significantly better than the Camel Case splitter ($d = 0.29$) and Samurai ($d = 0.22$).
- When using the program-level dictionary, TIDIER performs significantly better than the alternative approaches, although the effect size is very small ($d < 0.2$).
- When using the program-level dictionary augmented with domain knowledge, TIDIER again performs significantly better than the alternative approaches, with a medium effect size ($d = 0.58$ for the Camel Case splitter and $d = 0.51$ for Samurai).

We can summarize the results for **RQ1** as follows: with the simple English dictionary, TIDIER performs worse than the alternative approaches. However, TIDIER outperforms other approaches when the simple English dictionary is augmented with domain knowledge or, with even better results, when it uses a program-level contextual dictionary augmented with domain knowledge.

Regarding **RQ2**, we conclude that there are two factors contributing to the increase of performance of TIDIER: augmenting the dictionary with domain knowledge, using a program-level contextual dictionary, or, to obtain the best performances, augmenting a program-level dictionary with domain knowledge.

A summary of the comparisons among the different configurations of TIDIER—depending on the dictionary adopted—and the other two approaches, *i.e.*, Camel Case and Samurai, is

Copyright © 2000 John Wiley & Sons, Ltd.
*Prepared using* smrauth.cls

reported in Table VI, where approaches with better performances are labelled with a higher number of "*". Table VI provides a quick overview of the approach performances; for a precise and quantitative comparison of the approaches, the reader should look at the other tables reporting descriptive statistics and results of statistical tests.

### 5.2. RQ3

To answer **RQ3**, we ran five times TIDIER on each of the 73 abbreviations with the English dictionary of 2,774 words. Out of the 73 abbreviations that TIDIER could potentially map to dictionary words, TIDIER produced a correct mapping for 35 of them, achieving an accuracy of 48%. Although this percentage is not high, to the best of our knowledge, TIDIER is the first and only approach able to deal with abbreviations.

The first block of Table VII shows examples of abbreviations that were correctly mapped by TIDIER to dictionary words. The table reports: (i) the abbreviations, (ii) the oracle, and (iii) the different mappings produced by TIDIER. The second block of Table VII shows examples of wrong mappings, such as those of *auth* into *author* while the correct mapping was *authenticate*) or of *dest* into *destroy* while the correct one was *destination*. Wrong mappings happen because TIDIER does not use semantic information and, thus, can generate mappings that are different from those in our oracle even though with a zero distance. Consistently with insights gained from **RQ1** and **RQ2**, wrong mappings suggest that domain-specific dictionaries can be useful to better support identifier splitting.

Out of the $73 - 35 = 38$ abbreviations not correctly mapped by TIDIER, there are 16 identifiers wrongly mapped and 22 identifiers for which TIDIER was not able to produce a mapping with a zero distance. Some of these cases are shown in the third block of Table VII, where the numbers in parentheses report the achieved minimum distances. For example, *addr* was mapped to *add* instead of *address* with distance two (trailing *r* removed), *arch* into *march* instead of *architecture* (leading *m* added) with distance one, and *def* into *prefix* instead of *define* with distance two (leading *p* and *r* added).

In conclusion, **RQ3** suggests that TIDIER is indeed able to deal with the abbreviations used to build identifiers and can map them into dictionary words in 48% of the abbreviations considered in our sample. We claim that this result is promising because alternative approaches are not able to deal with abbreviations at all and because future work could improve the mappings, possibly using enhanced search heuristics.

### 5.3. General Discussions

Results presented here are valid for programming languages such as C or C++ where developers tend to use abbreviations and do not always use a consistent naming convention, *e.g.*, Camel Case or underscore, to separate words composing identifiers.

Our experience with Java is that developers tend to apply more consistently naming conventions than C and C++ developers. We conjecture that, for Java programs, the Camel Case splitter and TIDIER would produce very close results. This conjecture is supported by our detailed comparison [MGD$^+$10] in which we compared a Camel Case splitter to TIDIER on 957 identifiers from JHotDraw. In fact, both the Camel Case splitter and TIDIER obtained high

Copyright © 2000 John Wiley & Sons, Ltd.

Table VII. Examples of correct and wrong abbreviations

Match with the Oracle

| Abbreviation | Oracle | Mapping 1 | Mapping 2 |
|---|---|---|---|
| arr | array | array | arrow |
| clr | clear | clear | color |
| curr | current | current – | |
| dev | device | device | – |
| div | division | dividend | divided |
| intern | internal | internal | – |
| len | length | length | lender |
| lng | long | long | language |
| mov | move | move | – |
| sec | security | security | secret |
| snd | sound | sound | sand |
| spec | specify | specify | specialize |
| str | string | string | strict |
| wrd | word | word | – |

Wrong Mappings

| Abbreviation | Oracle | Mapping 1 | Mapping 2 |
|---|---|---|---|
| auth | authenticate | author | |
| comm | communication | comment | command |
| del | delete | deal | delay |
| dest | destination | destroy | |
| disp | display | dispatch | |
| exp | expresion | expansion | expire |
| mem | memory | membrane | memo |
| procs | process | protocol css | prototype css |
| vol | volume | voltage | voluntary |

Distance > 0

| Abbreviation | Oracle | Mapping 1 | Mapping 2 |
|---|---|---|---|
| acct | accounting | act (1.0) | |
| addr | address | add (1.0) | |
| arch | architecture | march (1.0) | |
| elt | element | felt (1.0) | |
| lang | language | long (2.0) | |
| num | number | enum (1.0) | |
| paren | parenthesis | green (3.0) | |

accuracy, about 93% and 96% respectively. On the same set of identifiers, Samurai achieved an accuracy similar to that of TIDIER with a substantially lower computation time.

In summary, sophisticated approaches for identifier splitting and expansion techniques, such as TIDIER or Samurai, are useful for programming languages, such as C or C++, where appropriate separators are not always used, and in contexts where developers tend to use abbreviations. These approaches are less useful when the Camel Case convention or some known separators are systematically used: then a simpler identifier splitting approach would suffice.

From a computational perspective, out of the three studied approaches, *i.e.*, Camel Case splitter Samurai, and TIDIER, TIDIER is the most computationally demanding; as discussed in Section 2.5. Indeed, TIDIER has a cubic distance evaluation cost plus the search time, while the Camel Case splitter is linear and Samurai is quadratic. The performance of TIDIER is highly dependent on the dictionary size and quality. In an extreme case, if each identifier is composed of dictionary words and split with an exact match, the complexity of TIDIER would be quadratic. In such a case, both the Camel Case algorithm and Samurai would perform equally well, depending on the program coding conventions.

From a practical perspective, understanding source-code identifiers using approaches such as the one proposed in this paper is a re-engineering activity usually performed off-line and, thus, we assume that any reasonable computation time would be acceptable. TIDIER, even with the largest dictionary among those considered, took few hours to split the 1,026 identifiers of our study. Clearly, if hundreds of thousands of identifiers must be processed, the current implementation of TIDIER is not suitable and heuristics must be used to reduce computation time. For example, identifiers consisting in single words contained in the dictionary, *e.g.*, `position`, and neither composed of multiple words nor containing abbreviations, could be filtered in linear time. Finally, TIDIER performances could be further improved by distributing the computation on multiple machines.

## 6. Threats to Validity

This section discusses threats to the validity of our study that could impact its results.

Threats to *construct validity* concern the relation between the theory and the observation. This threat is mainly due to mistakes in the oracle. We cannot exclude that errors are present in the oracle. As the intent of the oracle is to explain identifiers semantics, we cannot exclude that some identifiers could have been split in different ways by the developers that originally created them. This problem is related to guessing the developers' intent and we can only hope that, given the program 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. To limit this threat, different sources of information, such as comments, context, and online documentation were used when producing the oracle.

Threats to *internal validity* concern any confounding factors that could have influenced our study results. In particular, these threats are due to the subjectivity of the manual building of the oracle and to the possible biases introduced by manually splitting identifiers. To limit this threat, the oracle was produced by two of the authors independently and inconsistencies in splitting/mapping to dictionary words were discussed. The size of the oracle was chosen large enough to ensure that even an error of a few percent would not dramatically affect the comparisons.

Threats to *Conclusion validity* concern the relations between the treatment and the outcome. Proper tests were performed to statistically reject the null hypotheses. In particular, we used non-parametric tests, which do not make any assumption on the underlying distributions of the data, and, specifically, a test appropriate for categorical data (the Fisher's exact test) and one for paired, ranked data (the Wilcoxon paired test). Also, we based our conclusions not only on the presence of significant differences but also on the presence of a practically relevant difference, estimated by means of an effect-size measure. Last, but not least, we dealt with problems related to performing multiple Fisher and Wilcoxon tests using the Holm's correction procedure.

Threats to *external validity* concern the possibility of generalizing our results. To make our results as generalizable as possible, we selected our sample of identifiers from a very large set of open-source projects. The size of our sample (1,026 composed identifiers) is comparable to the one used by Enslen *et al.* in their work [EHPVS09]. Differently from our previous work [MGD$^{+}$10], we only consider C programs rather than Java program because Java identifiers are mostly built using the Camel Case convention and, quite often, using complete English words rather than abbreviations. Instead, the usage of a more complex splitting algorithm is particularly useful for the C programming language. Despite the sample size, we cannot exclude that performances would vary on other projects, *e.g.*, commercial source code, and programming languages.

Finally, as neither the Camel Case splitter nor Samurai are designed to expand contractions or to deal with common abbreviations, precision and recall favor TIDIER. However, in our oracle, acronyms and abbreviations are never expanded beyond obvious cases (*e.g.*, "src" for "source" or "img" for "image"). Thus, integrating dictionaries containing acronyms in some advanced Camel Case splitters or in Samurai would reduce the advantage of TIDIER. For example, it could be possible to integrate into Samurai other frequency tables (*e.g.*, of acronyms or known abbreviations) and its results would then most likely be comparable with those of TIDIER.

## 7. Related Work

The important role of identifiers in program comprehension, traceability recovery, feature and concept location motivates the large body of relevant work. In the following, we focus on the most relevant contributions to identifier splitting.

Takang *et al.* [TGM96] empirically analyzed the role played by identifiers and comments on source code comprehensibility. They conducted experiments to compare abbreviated identifiers to full-word identifiers and uncommented code to commented code. The results showed that commented programs are more understandable than non-commented programs and programs containing full-word identifiers are more understandable than those with abbreviated identifiers. Similar results have also been achieved by Lawrie *et al.* [LMFB07, LMFB06], who presented an approach named QALP (Quality Assessment using Language Processing) relying on the textual similarity between related software artifacts to assess software quality. The QALP tool leverages identifiers and related comments to characterize the quality of a program. Their empirical study conducted with 100 programmers showed that full words as well as recognizable abbreviations led to better comprehension. These results suggest that the identification of words composing identifiers, and, thus, of the domain concepts associated with them could contribute to a better comprehension.

Copyright © 2000 John Wiley & Sons, Ltd.
*Prepared using* smrauth.cls

Binkley *et al.* [BDLM09] investigated the use of different identifier separators in program understanding. They found that the Camel Case convention led to better understanding than underscores and, when subjects are properly trained, that subjects performed faster with identifiers built using the Camel Case convention rather than with underscores.

Other works [CT00, MMM03] have investigated the information carried by the terms composing identifiers, their syntactic structure and quality. The existence of "hard terms" that encode core concepts into identifiers was the main outcome of the study by Anquetil *et al.* [AL98]. An in-depth analysis of the internal structure of identifier was conducted by Caprile and Tonella [CT99]. They reported that identifiers are chosen to convey relevant information about the role and properties of the program entities that they label. They also observed that identifiers are often the starting point for program comprehension, especially when high-level views, such as call graph, are available.

Guidelines for the production of high-quality identifiers have been provided by Deißenböck *et al.* [DP05]. Methods related to identifier refactoring were proposed by Caprile and Tonella [CT00] and Demeyer *et al.* [DDN00].

De Lucia *et al.* [LDOZ06, DDO10] proposed COCONUT, a tool highlighting to developers the similarity between source code identifiers and comments and words in high-level artifacts. They empirically showed that this tool is helpful to improve the overall quality of identifiers and comments.

Some studies [ACC$^+$02b, MACHH05, MM03] reported the use of identifiers to recover traceability links. Also, textual similarity between methods within a class, or among methods belonging to different classes, has been used to define new measures of cohesion and coupling, *i.e.*, the Conceptual Cohesion of Classes proposed by Marcus *et al.* [MPF08b] and the Conceptual Coupling of Classes proposed by Poshyvanyk *et al.* [PM06], which bring information complementary to structural cohesion and coupling measures.

Antoniol *et al.* [ACC$^+$02a] observed that most of the application-domain knowledge that developers possess when writing code is captured by identifier mnemonics. They wrote that "[p]rogrammers tend to process application-domain knowledge in a consistent way when writing code: program item names of different code regions related to a given text document are likely to be, if not the same, at least very similar". Thus, how readily the semantics inherent to identifiers can be extracted is of key importance.

De Lucia *et al.* [DFOT07] used LSI to identify cases of low similarity between artifacts previously traced by software engineers. Their technique relied on the use of textual similarity to perform an off-line quality assessment of both source code and documentation, with the objective of guiding a software quality review process because the lack of textual similarity may be an indicator of low quality of traceability links. In fact, poor textual description in high-level artifacts or meaningless identifiers or poor comments in source code may point to a poor development process and unreliable traceability links.

Abebe *et al.* [AHM$^+$08] analyzed how the source code vocabulary changes during evolution. They performed an exploratory study of the evolution of two large open-source programs. The authors observed that the vocabulary and the size of a program tend to evolve the same way and that the evolution of the source code vocabulary does not follow a trivial pattern. Their work was motivated by the importance of having meaningful identifiers and comments, consistent with high-level artifacts and with the domain vocabulary during the life of a program.

Many commonalities can be found with previous work using the Camel Case convention to split identifiers and, in particular, with the work of Enslen *et al.* [EHPVS09], which we detailed in Section 3. We share with these previous works the goal of automatically splitting identifiers into component terms. However, we do assume the use of neither the Camel Case convention nor a set of known prefixes or suffixes. In addition, our approach automatically generates a thesaurus of abbreviations using transformation rules attempting to mimic the developers' cognitive processes when building identifiers.

## 8. Conclusion

The existing literature reports evidence for the importance of properly choosing identifiers for program comprehension and maintainability [CT99, CT00, LMFB07, LMFB06, TGM96]. In this context, we presented TIDIER, an approach to split identifiers into component terms inspired from speech recognition, using Dynamic Time Warping, a string-edit distance, and a hill-climbing search technique.

TIDIER finds the minimum edit distance between the identifier terms and dictionary words. It can split identifiers even in the absence of explicit separator (*e.g.*, underscore or Camel Case convention) and deals with abbreviations in identifiers. Therefore, it can map terms in identifiers to dictionary words and, thus, help the mapping of identifiers to domain concepts. Furthermore, it can be useful to better assess the quality of identifiers or to identify identifier refactorings.

TIDIER takes as input a thesaurus and an identifier to be split. It splits the identifier into terms that achieve the overall, minimum edit distance with respect to words (and their abbreviations) contained in the thesaurus. Abbreviations not present in the thesaurus are generated automatically using word transformation rules, mimicking developers' identifier creation processes.

To quantify the performances of TIDIER, we applied it to split a set of 1,026 C identifiers randomly extracted from a corpus of 340 open source programs. The 1,026 identifiers were manually split into terms to build an oracle against which TIDIER and two other approaches, a simple Camel Case splitter and Samurai [EHPVS09], were compared.

Reported results show that, with program-level dictionaries augmented with domain knowledge, *i.e.*, common acronyms, abbreviations, and C library functions, TIDIER significantly outperforms previous approaches. Specifically, TIDIER achieved with the program-level dictionary complemented with domain knowledge 54% of correct splits, compared to 30% for the Camel Case splitter and 31% for Samurai. Moreover, TIDIER was also able to map identifiers terms to dictionary words with a precision of 48% for a set of 73 abbreviations present in the oracle.

Future works will be devoted to improve the string-edit distance guiding TIDIER, to speed up the algorithm by reducing the search space, and to use semantic information. In fact, two of the main limitations of TIDIER are its pure lexical-level matching and its cubic complexity. We also plan to combine TIDIER with Samurai to benefit from their respective advantages while reducing their limitations.

## Acknowledgement

The authors are grateful and thank warmly Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker for sharing with us the Samurai frequencies tables. This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chairs in Software Evolution and in Software Patterns and Patterns of Software) and by G. Antoniol Individual Discovery Grant.

## REFERENCES

[ACC+02a] . Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[ACC+02b] . Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28:970–983, Oct 2002.

[AHM+08] . Surafel L. Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella, and Giuliano Antoniol. Analyzing the evolution of the source code vocabulary. In *Proceedings of 12th European Conference on Software Maintenance and Reengineering*, pages 189–198, Kaiserslautern, Germany, 2008. IEEE CS Press.

[AL98] . Nicolas Anquetil and Timothy Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of CASCON*, pages 213–222, December 1998.

[BDLM09] . David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, pages 158–167. IEEE Computer Society, 2009.

[CCD09] . Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Tracking your changes: a language-independent approach. *IEEE Software*, 27(1):50–57, 2009.

[CLR90] . Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introductions to Algorithms.* MIT Press, 1990.

[Coh88] . Jacob Cohen. *Statistical power analysis for the behavioral sciences.* Lawrence Earlbaum Associates, 2nd edition edition, 1988.

[CT99] . Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pages 112–122, Atlanta Georgia USA, October 1999.

[CT00] . Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 97–107, 2000.

[DDN00] . Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 166–177. ACM Press, 2000.

[DDO10] . Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Trans. Software Eng.*, (to appear), 2010.

[DFOT07] . Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), 2007.

[DP05] . Florian Deissenbock and Markus Pizka. Concise and consistent naming. In *Proc. of the International Workshop on Program Comprehension (IWPC)*, May 2005.

[EHPVS09] . Eric Enslen, Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009, Vancouver, BC, Canada, May 16-17, 2009*, pages 71–80, 2009.

[FWG07] . Beat Fluri, Michael Wursch, and Harald 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)*, pages 70–79, 2007.

Copyright © 2000 John Wiley & Sons, Ltd.

[Hol79] . Sture Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979.

[JH06] . Zhen Ming Jiang and Ahmed E. Hassan. Examining the evolution of code comments in postgresql. In *Proceedings of the 2006 International Workshop on Mining Software Repositories MSR 2006*, pages 179–180, 2006.

[LDOZ06] . Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Francesco Zurolo. Improving comprehensibility of source code via traceability information: a controlled experiment. In *Proceedings of 14th IEEE International Conference on Program Comprehension*, pages 317–326, Athens Greece, 2006. IEEE CS Press.

[Lev66] . Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, (10):707–710, 1966.

[LFB06] . Dawn Lawrie, Henry Feild, and David Binkley. Syntactic identifier conciseness and consistency. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation Philadelphia Pennsylvania USA*, pages 139–148, Sept 27-29 2006.

[LMFB06] . Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *Proceedings of 14th IEEE International Conference on Program Comprehension*, pages 3–12, Athens, Greece, 2006. IEEE CS Press.

[LMFB07] . Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[MACHH05]. Jonathan I. Maletic, Giuliano Antoniol, Jane Cleland-Huang, and Jane Huffman Hayes. 3rd international workshop on traceability in emerging forms of software engineering (TEFSE2005). In *ASE*, page 462, 2005.

[MF04] . Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics - (2nd edition)*. Springer-Verlag, Berlin Germany, 2004.

[MGD$^+$10] . Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), March 15-18 2010, Madrid, Spain*. IEEE CS Press, 2010.

[MM03] . Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the International Conference on Software Engineering*, pages 125–137, 2003.

[MMM03] . Ettore Merlo, Ian McAdam, and Renato De Mori. Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance*, 15(4):205–244, 2003.

[MPF08a] . Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[MPF08b] . Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Softw. Eng.*, 34(2):287–300, 2008.

[Ney84] . Herman Ney. The use of a one-stage dynamic programming algorithm for connected word recognition. *IEEE Transactions on Acoustics Speech and Signal Processing*, 32(2):263–271, Apr 1984.

[PM06] . Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of 22nd IEEE International Conference on Software Maintenance*, pages 469 – 478, Philadelphia Pennsylvania USA, 2006. IEEE CS Press.

[SC78] . Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics Speech and Signal Processing*, 26(1):43–49, Feb 1978.

[She07] . David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[TCAD10] . Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, Jan 2010.

[TGM96] . Armstrong Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Program Languages*, 4(3):143–167, 1996.

Copyright © 2000 John Wiley & Sons, Ltd.

Table VIII. Descriptive statistics of precision.

| Method | Dictionary | 1Q | Median | 3Q | Mean | $\sigma$ |
|---|---|---|---|---|---|---|
| CamelCase | | 0.00 | 0.50 | 1.00 | 0.45 | 0.44 |
| Samurai | | 0.00 | 0.50 | 1.00 | 0.50 | 0.43 |
| TIDIER | English dictionary | 0.00 | 0.25 | 0.67 | 0.38 | 0.41 |
| | English dict. + domain kn. | 0.25 | 0.50 | 1.00 | 0.58 | 0.40 |
| | WordNet | 0.00 | 0.50 | 1.00 | 0.43 | 0.41 |
| | Function | 0.00 | 0.00 | 0.00 | 0.14 | 0.28 |
| | File | 0.00 | 0.00 | 0.50 | 0.30 | 0.37 |
| | Application | 0.00 | 0.50 | 1.00 | 0.51 | 0.40 |
| | Application + domain kn. | 0.50 | 1.00 | 1.00 | 0.72 | 0.37 |

Table IX. Descriptive statistics of recall.

| Method | Dictionary | 1Q | Median | 3Q | Mean | $\sigma$ |
|---|---|---|---|---|---|---|
| CamelCase | | 0.00 | 0.50 | 1.00 | 0.44 | 0.44 |
| Samurai | | 0.00 | 0.50 | 1.00 | 0.50 | 0.43 |
| TIDIER | English dictionary | 0.00 | 0.33 | 1.00 | 0.40 | 0.42 |
| | English dict. + domain kn. | 0.33 | 0.67 | 1.00 | 0.64 | 0.39 |
| | WordNet | 0.00 | 0.50 | 1.00 | 0.45 | 0.41 |
| | Function | 0.00 | 0.00 | 0.00 | 0.14 | 0.29 |
| | File | 0.00 | 0.00 | 0.60 | 0.33 | 0.39 |
| | Application | 0.00 | 0.50 | 1.00 | 0.55 | 0.41 |
| | Application + domain kn. | 0.50 | 1.00 | 1.00 | 0.75 | 0.36 |

## APPENDIX – Additional Tables

Copyright © 2000 John Wiley & Sons, Ltd.