Computer Science and Information Systems 11(4):1191–1207 DOI: 10.2298/CSIS131005027C

DMOSS: Open Source Software Documentation Assessment

Nuno Ramos Carvalho[1], Alberto Simões[2], and José João Almeida[1]

[1] Department of Informatics, University of Minho
{narcarvalho,jj}@di.uminho.pt
[2] Centre for Humanistic Studies, University of Minho
ambs@ilch.uminho.pt

Abstract. Besides source code, the fundamental source of information about open source software lies in documentation, and other non source code files, like *README*, *INSTALL*, or *How-To* files, commonly available in the software ecosystem. These documents, written in natural language, provide valuable information during the software development stage, but also in future maintenance and evolution tasks.
DMOSS[3] is a toolkit designed to systematically assess the quality of non source code content found in software packages. The toolkit handles a package as an attribute tree, and performs several tree traverse algorithms through a set of plugins, specialized in retrieving specific metrics from text, gathering information about the software. These metrics are later used to infer knowledge about the software, and composed together to build reports that assess the quality of specific features.
This paper discusses the motivations for this work, continues with a description of the toolkit implementation and design goals. This is followed by an example of its usage to process a software package, and the produced report.

Keywords: open source, documentation analysis, data mining.

1. Introduction

Open source software wide spread adoption, including in the industry, has raised increased concerns related with software quality and certification [11]. In this context, the CROSS research project[4] aims at developing software analysis techniques that can be combined to assess open source software projects. Although most of the effort is invested analyzing source code, non-source code content found in packages can have a direct impact on the overall quality of the software. For example documentation, installation procedures, practical information available in *README* files etc. The goal of the DMOSS toolkit is to provide a systematic approach to gather metrics about this content and assess its quality. It starts by collecting the content written in natural language, processing it to compute metrics, and finally reasoning about these metrics to draw conclusions.

Documentation analysis is also relevant in other research areas. Program Comprehension (PC) is an area of Software Engineering concerned with gathering information and providing knowledge about software, to help programmers understand how a program

[3] Documentation Mining Open Source Software
[4] An infrastructure for Certification and Re-engineering of Open Source Software: `http://di.uminho.pt/cross` (Last accessed 14-02-2014).

works, to ease software evolution and maintenance tasks [24]. Many of the techniques and methods used rely on mappings between program elements and the real world concepts these elements are addressing [27]. Non-source code content included in software packages can provide clues and valuable information (e.g. [3,31,23]). Also, software maintainers often rely on documentation to understand some key aspects of the source code [30].

Assessing software quality for any given definition of quality is not easy [18] mainly due to subjectivity. The toolkit described in this works evaluates the non-source code files included in a software package. This set of files can include *README* files, *INSTALL* files, HTML (HyperText Markup Language) documentation pages, or even UNIX man(ual) pages. Instead of trying to come up with a general definition for quality, we select three main traits that we are concerned about. We envisage that these characteristics have a direct impact in the overall documentation quality regardless of the degree of individual subjectivity.

- Readability: text readability can be subjective, but there are linguistic characteristics that generally make it harder to read. Some of them can even be measured, as for example, the number of syntax errors or the excessive use of abbreviations.
- Actuality: this is an important feature of documentation and other textual files, they should be up-to-date, and refer to the latest version of the software.
- Completeness: this trait tells us how much the documentation is complete, and if it addresses all the required topics.

DMOSS processes a software package to gather information about specific metrics that are related with these traits. Reasoning about these metrics helps drawing conclusions relevant to assess the overall described traits. Based on these conclusions quantitative measures can be calculated about the quality of the non-source code content. Currently available tools in the framework, do not provide enough data to fully assess any of these traits, but new tools can be added to increase analysis coverage of these traits.

The next section of this article discusses some related work in this area. Section 3 introduces the DMOSS toolkit and gives an overview about its implementation details. This section also illustrates the major algorithms used. Section 4 presents a quick tour about using the toolkit, and examples of generated reports and discussion are presented in Section 5. Finally, Section 6 concludes with some final remarks and discusses some trends for future work.

During the remainder of this paper the software package $tree$ [5] (version $1.5.3$) is used for illustration purposes, mainly because it is small and produces outputs that can fit in the paper size without jeopardizing reading.

2. Related Work

Software Measurement is an active research field. Buglionie and Abran [6] discuss how software measurement should be considered a knowledge area for software engineering,

[5] Available from: `http://mama.indstate.edu/users/ice/tree/` (Last accessed: 12-02-2014).

or if it should be considered orthogonal and part of every other. Curiously, the authors neither refer to documentation quality measurement, nor any of the knowledge areas previously defined support software documentation as part of the software development. Khelifi and Abran [16] discuss a standard approach to software measurement, suggesting a final report on the software measurement results, but this report is the only documentation ever referred. It even gets more disappointing when books on software metrology [1] do not refer documentation at any point.

Nevertheless, there are some researchers worrying about the status of software documentation, and how software engineers use the documentation. Lethbridge *et al* [21] do an overview on the status of software documentation, pointing out their main problems: being out-of-date; badly written; too much documentation that is mandated, time consuming to write, and mostly not useful. The authors also test the documentation usefulness, but do not specify any metrics to assess documentation quality. Also related to the documentation quantity (and not quality), Briand [5] discusses the issue of documentation quantity and documentation type, namely focusing on the use of the Unified Modeling Language (UML).

Souza *et al.* [29] do an interesting job on analyzing what are the documentation needs, using a pair of surveys targeting software maintainers, one asking to rate the importance of the documentation artifacts in helping to understand a system, and a second one to indicate which documentation artifacts they had used to gain understanding of a software they just finished maintaining. The results were interesting: for the first survey software maintainers rate source code as the primary source of documentation, and comments as the second. For the second survey, source code remains in the first position but comments drop to the third place, giving the place to unitary tests. Although these surveys do not help on measuring automatically the quality of software documentation it points on which artifacts should be analyzed.

Forward *et al.*, in their survey about the general opinion of software professionals regarding the relevance of documentation and related tools [10], highlight the general consensus that documentation content is relevant and important. They also highlight a set of concerns that software documentation technologies should be more aware of professionals' requirements, opposed to blindly enforce documentation formats or tools.

Scacchi, in his work about the requirements for open source software development [28], highlights not just the relevance of system documentation, but also the relevance of informal documents (for example, *How-Tos*). They are significant not only for documenting the system itself, but also to communicate important information for other people in the community (for example, how to contribute for the project).

There is a substantial body of work which illustrates the relevance of documentation quality in the context of software development and maintenance. Chen *et al.* [7] have identified that documentation quality is a dimension by itself, and a key problem factor that affects software maintenance phase.

Nevertheless, the literature is sparse when describing metrics and methods for evaluating non-source code content for software quality assessment. This work focuses on addressing this problem. Other researches have also been motivated to address text quality measuring in other contexts. Anderka *et al.* [2] devised a method for detecting text quality flaws as a classification problem. Dalip *et al.* [8] introduce a method, based on regression analysis, to aggregate several text quality indicators. Both these approaches

can be almost directly implemented as DMOSS plugins, and provide features to assess documentation quality, mainly regarding the readability trait. Methods related with text assessment used in several areas can be used to provide more metrics about package non-source content, for example: machine translation evaluation methods (e.g. [26]), students textual answer assessment (e.g. [25,15]), summaries automatic evaluation (e.g. [22,12]), or web content assessment (e.g. [17]).

3. DMOSS Toolkit

The DMOSS toolkit's main goal is to provide a set of tools that systematically process a software package, and produce a report with conclusions about the quality of the non-source code content found. This includes analyzing all the natural language text available in the documentation, comments in the code, and other non-source code files typically found in packages.

The main design goals for DMOSS are:

- Develop small autonomous tools, so that they can be useful in other contexts or environments. Applications that use these smaller tools are modular, so that new tools can be added without any additional effort, just like typical plugins.
- Many tools in DMOSS take advantage of known algorithms and techniques (for example the file *processors*). The main engine in the toolkit needs to be based on the usage of plugins, so that new processors and similar utilities can be added and improved easily.
- Represent the software package as a tree. This allows the implementation of the analysis algorithms as a set of tree traversals (for more details on tree data structures and traversal algorithms see e.g. [9,19]). Keeping the implementation of the specific analysis algorithms self-contained in the plugins.

Let us stress again the importance regarding the way DMOSS represents a software package: an annotated tree. In this tree, nodes represent files and directories, and edges describe the hierarchical structure of the package. An example tree is illustrated in Fig. 1, for the $tree$ software package. This tree is automatically generated by the toolkit, by traversing the filesystem file hierarchy recursively, adding nodes for each file and directory visited. For each node that is a known documentation format (e.g. man(ual) page, HTML) the plain text content is extracted and added to the corresponding node as an attribute [6].

Once this tree is available, the task of processing a software package is divided in two tree traversals:

1. During the first pass the goal is to gather information about files and their content. Each plugin *processor* function is executed for each file individually, and the computed metrics are stored in the tree as node attributes.
2. In the second pass, results are aggregated (or reduced). For each directory node, the available features are reduced to a single result. In the end, the tree root node (the package top directory) contains the results of processing the entire package.

[6] Known types are defined in the toolkit by a set of regular expressions that match file extensions, and a dispatch table that contains functions definitions for text extraction for each type.

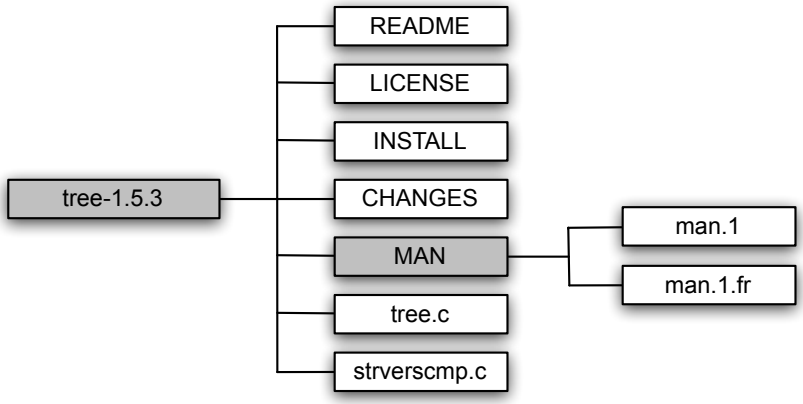


Fig. 1: DMOSS software package tree like structure representation.

After these traversals, a final report with conclusions is produced using the data stored in the annotated tree. The plugins that perform the actual analysis and build conclusions need to implement three functions to be used in both tree traversals:

1. A *processor*, which is responsible for gathering information about a file and produce a set of features (a metric can be measured using one or more features) about its content. These features are stored in the tree as node attributes:

$$processor :: Node \longrightarrow [Feature]$$

2. A *reducer*, which is responsible for reducing features to produce either intermediate or final results. Results can be a single feature or a set of features:

$$reducer :: [Feature] \longrightarrow [Feature]$$

3. Finally, a *reporter*, which is responsible for building the final report given a set of features:

$$reporter :: [Feature] \longrightarrow Report$$

The only strictly required function is the *processor*, as there are default implementations for the other two functions, which are used when a plugin does not provide them. The default *reducer* reduces attributes using string concatenation or arithmetic sum depending on value type. The default *reporter* uses a pre-defined template to produce a simple report.

A feature is defined as a pair, consisting of a name and a value:

$$Feature = Name \times Value$$

where,

– $Name$ is the attribute identifier (a string);

– $Value$ can be an atomic value (a string or number for example), or a structured set of more $Feature$s for storing complex data structures.

A node in the tree is defined as:

$$Node = Path \times isFile \times Text \times Features$$

where,

– $Path$ stores the file name and its path;
– $isFile$ is a boolean value stating if this node is a file or a directory;
– $Text$ stores the natural language text found in the file., and is computed before starting the tree traversal stages. During this step, content is extracted from files written in known formats (e.g. HTML, POD, *man*) and stored in the tree as plain text.
– $Features$ stores a set of features for each node.

3.1. First Pass: Gathering Information

When traversing the tree, each file node is processed, i.e. the files represented by each node are processed. These nodes are processed in two steps:

1. Determine the file type, either using its full media type [13], or using heuristics, like the file header or extension. The result of this step is the creation of an attribute named *type* with the corresponding file type (for example *plain/text*, *text/xml* or *text/html*) as its value.
2. Given the node *type* and a list of available *processors* for each file type[7] the next step is to process the current file with all the available processors that support it, and store each processor resulting feature as a new node attribute.

This workflow is executed for every single node that represents a file, and is illustrated in Algorithm 1 [8]. The final result is a tree with a set of metrics calculated for each file node, and stored as attributes (including the file type).

Processors compute attributes values for file nodes, the toolkit provides an heterogeneous set of processors. Each processor typically handles a single file, and produces a result that is stored as an attribute in the tree. For example, the spell checker processor computes the total number of words in a text file, and the total number of words found in the dictionary (see Algorithm 2), the dictionary used is $aspell$ [9].

New processors can be added or *plugged in* at any time. Each plugin is also responsible for defining which file types it wants to process. This information is used to build a dispatch table before any transversal, which keeps the traversing tree engine agnostic to which processors are available, and which files to process.

[7] The toolkit provides a set of plugins that implement several *processors* (more details in Section 3.4), and new plugins can be easily added.

[8] Algorithm 1 and 3 loop over the relevant nodes in the tree are simplified for illustration purposes, the actual implementation follows the traditional tree traversal algorithms described in the literature.

[9] Available from: `http://aspell.net` (Last accessed: 12-02-2014).

Algorithm 1: Decorate tree with *processor*s results

Input: tree : Tree representing package content.
Input: processors : Set of processors indexed by type.
Result: Tree after adding processors resulting features.
foreach $node \in tree : node.isFile = True$ **do**
  $type \leftarrow typeOf(node)$ *// compute file type*
  foreach $processor \in processors(type)$ **do**
    $node.features.push(processor(node))$ *// add resulting feature set to node*
return $tree$

Algorithm 2: Processor example: Spell Checker

Input: node : Node representing the file being processed.
Result: New feature set to be added to the node.
$total \leftarrow 0$
$found \leftarrow 0$
foreach $word \in split\_words(node.text)$ **do**
  if $dictionary.valid(word)$ **then**
    $found \leftarrow found + 1$ *// word was found in the dictionary*
  $total \leftarrow total + 1$
$f1 \leftarrow Feature$("spellCheckerTotal", $total$)
$f2 \leftarrow Feature$("spellCheckerFound", $found$)
return $[f1, f2]$

3.2. Second Pass: Reducing Results

The goal of the second tree traversal (depth-first [20]) is to produce the final feature set. This is achieved by combining (or reducing) the intermediate results for every level of the package tree, and adding new attributes (typically to the directories nodes) that store the result of combining the features for each subtree. Every plugin may provide a specific function to combine results. The default method for combining intermediate results is plain string concatenation, or arithmetic addition (depending on value type).

For example, the combining function for the spell checker processor is to add the total number of words, and the total number of words not found for the files on each directory. This means that after this pass, the `MAN` node (illustrated in Figure 1, which represents the file-system `man/` directory) has an attribute that stores the result of combining the spell checker processor result for files `man.1` and `man.1.fr`[10]. Later, this attribute value is used to calculate the totals for the package, stored in the top level directory.

Figure 2 illustrates this process for an arbitrary metric. The algorithm is also described in Algorithm 3.

Reducers are used to reduce intermediate results, *i.e.*, combine the results found by the processors in the subtree of the node currently being processed, and add this reduced

[10] Although the two files are written in different languages the plugin uses a language identification algorithm before the spell checking task.

Algorithm 3: Transversing the annotated tree, depth-first, to reduce nodes features

Input: tree : Tree representing package content.
Input: reducers : Set of available reducers indexed by feature.
Result: Tree after adding reducers results to nodes as features.
foreach $node \in transverse\_depth\_first(tree) : node.isFile = False$ **do**
  foreach $reduce \in reducers$ **do**
    $features \leftarrow ...$ *// get features from children node set*
    $result \leftarrow reduce(features)$ *// call reduce function*
    $node.features.push(result)$ *// add reducing result features to node*
return $tree$

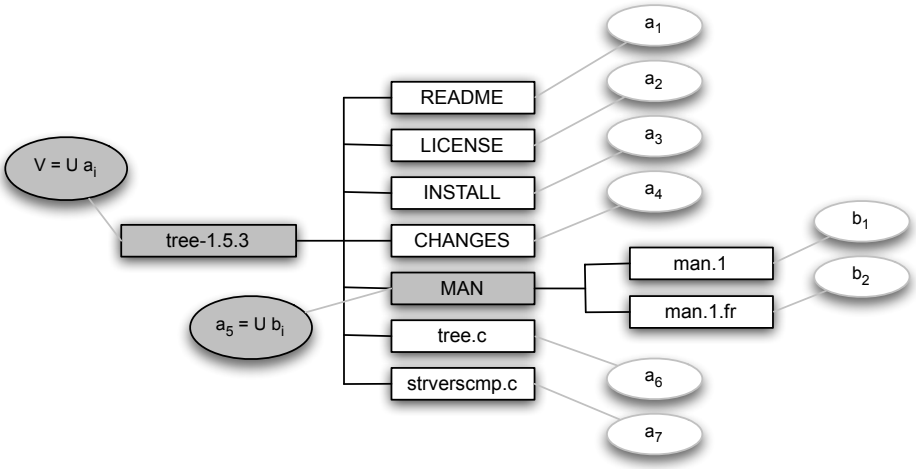


Fig. 2: Calculate final values recursively.

result to the current node as new features. Algorithm 4 illustrates the reducer for the spell checker example.

3.3. Building Reports

After the package is processed, a tree representing the package is available. This tree is decorated with a set of features per node, that convey all the results gathered from processing each file node, and also the conclusions taken for each processor. This information is stored in the tree using attributes. The toolkit provides a tool that can build reports in several formats including HTML, and ontology style graphs in GraphViz[11] notation. Examples of a HTML formatted reports are illustrated in Figure 3.

After the tree traversal stages, the set of reporters functions can be used to produce a final report. In this step all the reporters functions are executed, and the results are aggregated to build the final report (Algorithm 5). Besides these structured reports, the

[11] Available from: `http://www.graphviz.org/` (Last accessed: 12-02-2014).

Algorithm 4: Reducer example: Spell Checker

Input: features : Set of features.
Result: New set of features to be added to the node.
$total \leftarrow 0$
$found \leftarrow 0$
foreach $feature \in features$ **do**
  if $feature.name =$ *"SpellCheckerTotal"* **then**
    $total \leftarrow total + feature.value$
  if $feature.name =$ *"SpellCheckerFound"* **then**
    $found \leftarrow found + feature.value$
$f1 \leftarrow Feature$("spellCheckerTotal", $total$)
$f2 \leftarrow Feature$("spellCheckerFound", $found$)
return $[f1, f2]$

Algorithm 5: Build final report

Input: tree : Tree representing package content.
Input: reporters : Set of available reporters.
Result: Final HTML report.
$report \leftarrow$ "" *// start with an empty report*
$features \leftarrow ...$ *// collect features set from tree root*
foreach $reporter \in reporters$ **do**
  $curr \leftarrow reporter(features)$ *// build each individual report*
  $report \leftarrow concat(report, curr)$ *// concatenate individual reports*
return $report$

full tree is available as an associative array to be further processed by any other tool or application.

Reporters process a specific set of features about the package and produce custom reports. They are mainly used for producing reports that require post processing computations to achieve the intended result in the report (averages computations, for example). Reporters usually compute a final grade for a specific analyzed feature (the formula for computing the grade is another responsibility of a reporter function). Reporters' output is usually a snippet of HTML, built using a default set of templates. The complete tree is always available inside any reporter function, to gather any required information to build a more detailed report.

3.4. Toolkit Plugins

This section gives a brief overview of the plugins currently included in the DMOSS toolkit, and used to produce the reports illustrated in the next section.

Validate Links gathers links found for known protocols (e.g. HTTP, FTP), and checks if the link is still working. To validate the link a simple request is made, and if a successful reply is received the link is considered valid. The plugin rates the package better, as more valid links are found.

Spell Checker performs word spell checking, using a general purpose dictionary, for every word found in the documentation, and other non-source code files. It automatically detect the language used, and chooses the dictionary accordingly. The dictionary engine used is $aspell$.

Verify Licenses gather possible license information found in the software package. It can verify some common open source license (e.g. GNU General Public License [12]) used in software packages. Since this plugin was initially created for open source software, it grades the package if two criteria are met: (1) license information is found, (2) a known open source license was found. Of course, more licenses can be added.

Comment Lines counts the total number of source code lines, and the total number of comment lines found in the package source files. While the number of comment lines per lines of source code is above 19% [13] this plugin grades the files positively.

Identifiers Found in Docs attempts to measure documentation source code coverage. It measures the number of program identifiers (strings used as functions or variables names) found in the documentation. Mainly because most documentation formats (e.g. DoxyGen) use these strings to relate the documentation snippets with the source code. This can help to have an idea of which source code is covered by documentation.

Automatic Classification is used to automatically classify the software package using SOURCEFORGE taxonomy [14]. The classification algorithm is straight-forward, it mea-

[12] General information about GNU licenses available from: `http://www.gnu.org/licenses/` (Last accessed: 12-02-2014).

[13] This particular threshold was chosen based on a study by Arafat *et al.* about source code comments practices in open source projects [4].

[14] Available from: `http://sourceforge.net/` (Last accessed: 12-02-2014).

sures the distance between the words found in the documentation (which are valid according to the english dictionary), and the terms in the taxonomy. A more robust version of this plugin should use a well established classification approach like Support Vector Machine (SVM) [14] or Naive Bayes related algorithms [32]. The correct automatic classification of the package can be a positive characteristic, because there's a good probability that the vocabulary used in the documentation is close with the vocabulary stored in the taxonomy index, which is usually in line with the vocabulary used in the software area of interest. This plugin grade is directly related with two factors: (1) classification was possible, and (2) the degree of confidence (distance) on the computed classification.

Changes Verification is used to analyze changes information, if available. This file typically describes major releases done for the software, including the date for the release, and a list of topics that describe the major changes. The current goal of this plugin is to discover the date of the last release, and compare it to the current year. This is used in the report to grade positively packages with more recent releases. The lower possible grade is given when the set of regular expressions that are used to parse the file content are not able to return any information. Although, there is no standard format for these files, this can be an indicator that maybe some of the best practices were not followed.

New plugins can be easily added to analyze other features or characteristics of the package. New plugins just need to define the required functions as described in the previous sections. The set of plugins available in DMOSS do not cover all the measures and metrics described in the literature, and new analysis are proposed every day. One of the major goals of the proposed methodology is to provide the community with a framework that allows the quick development of new measurements, and integration with currently available ones.

3.5. Traits *Versus* Plugins

In line with the previous discussion, the described plugins only provide information to assess a limited number of features. These features are related with previously discussed traits, that tend to have a considerable weight in the overall package quality.

Plugins like *Spell Checker* are close related to readability, the increasing number of spelling errors can introduce noise in the text, making it harder to read or understand. Other features that can be measured to increase readability coverage are, for example, excessive use of acronyms and abbreviations, or punctuation analysis.

Completeness is a trait concerned with how much of the source code, and related concepts, are covered by documentation. The *Comment Lines* plugin measures the ratio of lines of comments found per lines of code, a low ratio may suggest that there are big portions of undocumented code. This feature is close related to the *Identifiers Found in Docs* plugin measure, by finding function definitions that lack corresponding documentation. This can be crucial when documentation generation systems are used (e.g. Doxygen). Both these traits are useful during software development, measures can be used during development stages to make sure documentation is keeping up with code implementation.

Plugins like *Validate Links* are more related to the actuality trait, because they provide clues that some elements in the documentation may be out of date, by finding links that

are no longer active or have been moved elsewhere. Another possible clue can be given by the *Changes Verification* plugin, if a software package is stalled in time, i.e has not released a new version in the last years, it might be prone to have outdated content.

Other plugins tend to be less subjective, and provide accurate information about a specific propriety, and are not closely related to these traits. For example the *Verify Licenses* plugin attempts to answer a specific question: *"Which license is the software package released under?"*. This is a relevant detail in the context of open source software.

4. DMOSS Quick Tour

This section illustrates a step-by-step usage of the toolkit applied to the `tree` software package.

The first step is to process the software package, this is done using the `dmoss-process` tool, which has a mandatory argument, either the file, or the complete URL for the package. The result of processing the given package is a tree, decorated with attributes storing the computed features, by default this tree is stored in a filename called `dmoss.data`. An example of execution is:

```
$ dmoss-process tree-1.5.3.tgz
Data saved as dmoss.data
```

This builds the tree representing the package, and executes all the tree traverses described in Section 3. This information can now be used by other tools, including the tool that builds a final report about the package, using all the defined reporters functions. An example of execution of this tool is:

```
$ dmoss-report dmoss.data > report.html
```

The result `report.html`, illustrated in Fig. 3c, shows metrics that are used to grade key features about the package. For example, many documents in software packages contain links to official websites or discussion forums, one of the plugins included in the toolkit validates that these link are still working. If all links included in the documentation are working this feature is graded positively. Another example is the number of comment lines in order to the total source code lines. In this specific case the percentage of comment lines per number of line codes is below 20%, which graded this feature of documentation with grade *F*. Some of these features are based on thresholds, that can be configured and adapted to specific contexts or packages. By clicking on each specific feature in the HTML report, more information is shown regarding each specific metric, as an example the detailed information about the spell checker is illustrated in Fig. 4b. Features are graded from *F* to *A*, being *A* the best grade, this information is included in the report in the form of icons, and is also used to choose the background color for each feature box.

The `dmoss.data` files stores the annotated tree, and all the information gathered during the processing stages. And can be used to implement other tools that build different outputs. The file is stored in a binary format, but the `dmoss-dump` tool (also included in the toolkit) can be used to print the data in a plain text format. An example of execution is:

```
$ dmoss-dump dmoss.data
```

This tool main purpose is to be used for debugging tasks. The output format is defined by `Data::Dumper` [15], where all the hash tables found in the data structure are represented using `{ }`, and lists using `[ ]`. Members of the hash tables are represented using the format: key => $value$, and list elements are separated by a `,` (comma), scalars values are printed normally, and references are represented using a concatenation of their type and identifier. More details about the format are described in the library documentation.

5. Discussion

To have an idea of DMOSS performance and effectiveness, some open source tools were processed, and the resulting reports were thoroughly reviewed. Figure 3 illustrates some final reports, from open source analyzed packages: (3a) $aspell\,0.60$, a spell checker; (3b) $wget\,1.9.1$ [16], a package for retrieving files using several protocols; (3c) $tree\,1.5.3$, a recursive directory listing command; and, (3d) $grep\,2.9$ [17], a tool for searching patterns in plain text files.

To measure the plugins effectiveness, several features were randomly selected for each packages, and were verified by hand. Most of the results reviewed were found to be correct from a measurement point of view, and inline why the expected result. The only plugin that raised questions during this process was *Automatic Classification*, that due to the simple algorithm adopted although the classification itself is almost always attained, the resulting class is not always the best choice.

Concerning the actual conclusions drawn by the reports, in general there is a pattern of features that are prone to lower grades. This pattern is visible in Fig. 3, the color red refers to lower grades, green to better grades, and yellow is in the middle. The ratio of comment lines per lines of code is a clear example of a measure that is prone to lower grades, all the reports show this plugin as red. Maybe the threshold used are in practice too demanding, or other sources of documentation (e.g. tool website) should also be included in this ratio. The *Changes Verification* is another plugin that is prone to lower grades, mainly because the lack of a common format to describe package changes. Also, the information might be available someplace else, which the plugin is not set to analyze.

On the other hand, the *Link Validator* plugin and the *Spell Checker* are more prone to have better grades (see the green colors in Fig. 3). The reasons for this are mainly: (1) its' easy for the programmer writing the documentation to verify this information, i.e. its' common practice to test a link before adding it to a document, or most currently available editors already integrate a spell checker – less spelling errors or typos – and, (2) this information is easier to verify, parsing documents for gathering links and words is a relatively easy task, so the bias introduced by the plugin processing stage in the final result is lower.

---

[15] A library to write complex data structures in plain text, available from: `http://search.cpan.org/dist/Data-Dumper/` (Last accessed: 12-02-2014).

[16] Available from: `http://www.gnu.org/software/wget/` (Last accessed: 12-02-2014).

[17] Available from: `http://www.gnu.org/software/grep/` (Last accessed: 12-02-2014).

(a) *aspell* 0.60 **(b)** *wget* 1.9.1

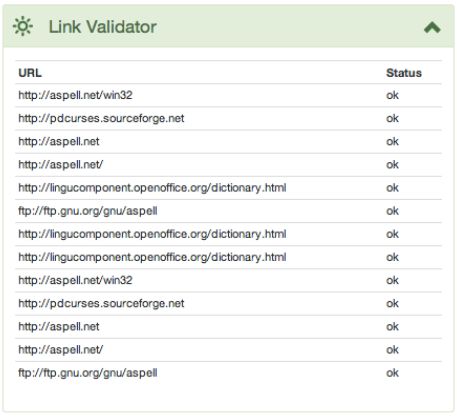
(c) *tree* 1.5.3 **(d)** *grep* 2.9

Fig. 3: Screenshots of HTML reports produced using DMOSS for several software packages, including analyzed features and corresponding grades.

6. Conclusion

Non-source code content found in software packages provides useful insight and information about the software and its' domain. This information can be used in distinct areas: software certification, or source code understanding for software maintenance or evolution. The major contribution of this work is a framework that can be used to quickly implement and integrate new metrics and analysis for documentation, and other non-source code content found in software packages, in order to assess its' quality.

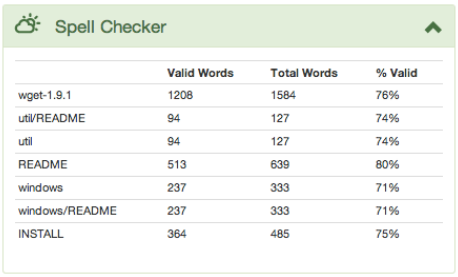
The DMOSS toolkit represents a software package as a tree, where nodes are used to represent files and directories. Each node includes a set of attributes, a list of pairs *key*/*value*, that store heterogeneous information. Using this representation we were able to implement the major package processing stages using traditional tree traversal algorithms, keeping the individual feature analysis self contained in plugins. This approach



Link Validator

URL	Status
http://aspell.net/win32	ok
http://pdcurses.sourceforge.net	ok
http://aspell.net	ok
http://aspell.net/	ok
http://lingucomponent.openoffice.org/dictionary.html	ok
ftp://ftp.gnu.org/gnu/aspell	ok
http://lingucomponent.openoffice.org/dictionary.html	ok
http://lingucomponent.openoffice.org/dictionary.html	ok
http://aspell.net/win32	ok
http://pdcurses.sourceforge.net	ok
http://aspell.net	ok
http://aspell.net/	ok
ftp://ftp.gnu.org/gnu/aspell	ok

(a) *aspell* 0.60 link validator details



Spell Checker

	Valid Words	Total Words	% Valid
wget-1.9.1	1208	1584	76%
util/README	94	127	74%
util	94	127	74%
README	513	639	80%
windows	237	333	71%
windows/README	237	333	71%
INSTALL	364	485	75%

(b) *wget* 1.9.1 spell checker analysis details.

Fig. 4: Each plugin provides detailed information about the analysis done, this can be accessed by clicking the relevant box.

has allowed the development of a modular and *pluggable* toolkit, easy to maintain and extend. The toolkit is platform independent, it can process any software package, regardless of programming languages used. The tool responsible for extracting plain text from arbitrary files may require update for some specific formats.

Regarding the obtained results from software package analysis, we noticed that there is a great concern about overall package natural text information content. Nowadays, open source communities spend time making sure that information for users and developers is available, and up-to-date. There is also a concern with information related with licenses and other *non software engineering* content. Some features are more prone to have lower grades than others, for example the number of comment lines per lines of code. Many of the plugins implemented use some kind of thresholds to grade some specific features, these may required some fine-tuning. Another example is the identifiers found in the documentation, packages may have use more abstract documentation, or use documentation generation tools, and this can jeopardize this analysis results.

Some tasks that can be developed in the future to improve this work:

- increase the number of available plugins, and thus increase the number of analyzed features;
- implement tools that provide other views of the decorated tree, for example browsable graphs;
- some key features require a more detailed investigation because they are prone to less grades, and maybe the evaluation process needs to be relaxed;
- mining data from external sources of information related with the software that live outside the package (official website or discussion *wikis* for example).

Acknowledgments. This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundaçao para a Ciencia e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010049

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

References

1. Abran, A.: Software Metrics and Software Metrology. John Wiley; Sons Interscience and IEEE-CS Press, New Jersey (2010)
2. Anderka, M., Stein, B., Lipka, N.: Towards automatic quality assurance in wikipedia. In: Proceedings of the 20th international conference companion on World wide web. pp. 5–6. ACM (2011)
3. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. Software Engineering, IEEE Transactions on 28(10), 970–983 (2002)
4. Arafat, O., Riehle, D.: The commenting practice of open source. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. pp. 857–864. ACM (2009)
5. Briand, L.: Software documentation: how much is enough? In: Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on. pp. 13–15 (2003)
6. Buglionie, L., Abran, A.: Software measurement body of knowledge – overview of empirical support. In: Proceedings of the 15th International Workshop on Software Measurement – IWSM 2005. pp. 353–368. Shaker Verlag, Montréal (Canada) (2005)
7. Chen, J., Huang, S.: An empirical analysis of the impact of software development problem factors on software maintainability. Journal of Systems and Software 82(6), 981–992 (2009)
8. Dalip, D.H., Gonçalves, M.A., Cristo, M., Calado, P.: Automatic assessment of document quality in web collaborative digital libraries. Journal of Data and Information Quality (JDIQ) 2(3), 14 (2011)
9. Ford, W., Topp, W., Ford, W.H.: Data Structures with C++ Using STL, 2/e. Pearson Education India (2002)
10. Forward, A., Lethbridge, T.: The relevance of software documentation, tools and technologies: a survey. In: Proceedings of the 2002 ACM symposium on Document engineering. pp. 26–33. ACM (2002)
11. Hauge, Ø., Ayala, C., Conradi, R.: Adoption of open source software in software-intensive organizations–a systematic literature review. Information and Software Technology 52(11), 1133–1154 (2010)
12. He, Y., Hui, S.C., Quan, T.T.: Automatic summary assessment for intelligent tutoring systems. Computers & Education 53(3), 890–899 (2009)
13. IANA: MIME Media Types. Web site: http://www.iana.org/assignments/media-types/index.html [Last accessed: 2012-11-27]
14. Joachims, T.: Text categorization with support vector machines: Learning with many relevant features. Springer (1998)
15. Kakkonen, T., Sutinen, E.: Automatic assessment of the content of essays based on course materials. In: Information Technology: Research and Education, 2004. ITRE 2004. 2nd International Conference on. pp. 126–130. IEEE (2004)
16. Khelifi, A., Abran, A.: Software measurement standard etalons: A design process. International Journal of Computers 1 (2007)
17. Kim, W., Aronson, A.R., Wilbur, W.J.: Automatic mesh term assignment and quality assessment. In: Proceedings of the AMIA Symposium. p. 319. American Medical Informatics Association (2001)
18. Kitchenham, B., Pfleeger, S.: Software quality: the elusive target [special issues section]. Software, IEEE 13(1), 12–21 (1996)

19. Knuth, D.E.: The art of computer programming, volume 2: seminumerical algorithms. Reading, Mass.: Addison-Wesley (1981)
20. Leiserson, C.E., Rivest, R.L., Stein, C., Cormen, T.H.: Introduction to algorithms. The MIT press (2001)
21. Lethbridge, T.C., Singer, J., Forward, A.: How software engineers use documentation: The state of the practice. IEEE Softw. 20(6), 35–39 (Nov 2003), http://dx.doi.org/10.1109/MS.2003.1241364
22. Lin, C.Y.: Rouge: A package for automatic evaluation of summaries. In: Text Summarization Branches Out: Proceedings of the ACL-04 Workshop. pp. 74–81 (2004)
23. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Software Engineering, 2003. Proceedings. 25th International Conference on. pp. 125–135. IEEE (2003)
24. Nelson, M.L.: A survey of reverse engineering and program comprehension. CoRR abs/cs/0503068 (2005)
25. Noorbehbahani, F., Kardan, A.A.: The automatic assessment of free text answers using a modified bleu algorithm. Computers & Education 56(2), 337–345 (2011)
26. Papineni, K., Roukos, S., Ward, T., Zhu, W.J.: Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting on association for computational linguistics. pp. 311–318. Association for Computational Linguistics (2002)
27. Rajlich, V., Wilde, N.: The role of concepts in program comprehension. In: Program Comprehension, 2002. Proceedings. 10th International Workshop on. pp. 271–278. IEEE (2002)
28. Scacchi, W.: Understanding the requirements for developing open source software systems. In: Software, IEE Proceedings-. vol. 149, pp. 24–39. IET (2002)
29. de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A study of the documentation essential to software maintenance. In: Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information. pp. 68–75. SIGDOC '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1085313.1085331
30. Thomas, B., Tilley, S.: Documentation for software engineers: what is needed to aid system understanding? In: Proceedings of the 19th annual international conference on Computer documentation. pp. 235–236. ACM (2001)
31. Yadla, S., Hayes, J.H., Dekhtyar, A.: Tracing requirements to defect reports: an application of information retrieval techniques. Innovations in Systems and Software Engineering 1(2), 116–124 (2005)
32. Yong-feng, S., Yan-ping, Z.: Comparison of text categorization algorithms. Wuhan university Journal of natural sciences 9(5), 798–804 (2004)

Nuno Ramos Carvalho is currently a PhD student in University of Minho. His main areas of research interests are the design and implementation of domain specific languages, construction of compilers and other language-based tools, software reverse engineering and Natural Language Processing.

Alberto Simões has a PhD in Natural Language Processing. His main research areas are Languages Processing. At the present he is an invited lecturer at University of Minho and Polytechnic Institute of Cavado and Ave, Portugal.

José João Almeida teaches in Department of Informatics, University of Minho in the area of compilers and Natural Language Processing.

Received: October 5, 2013; Accepted: May 8, 2014.