Open Source Software Documentation Mining for Quality Assessment

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

[1] Departamento de Informática, Universidade do Minho
{narcarvalho,jj}@di.uminho.pt
[2] Centro de Estudos Humanísticos, Universidade do 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 *HowTo* 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 text 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 of the software. This paper discusses the motivations for this work, continues with a description of the toolkit implementation and design goals. Follows an example of its usage to process a software package, and the produced report. Finally some final remarks and trends for future work are presented.

1 Introduction

Open Source Software (OOS) wide spread adoption, including in the industry, has raised increased concerns related with software quality and certification [3]. 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 spent 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

[3] Documentation Mining Open Source Software

[4] An Infrastructure for Certification and Re-engineering of Open Source Software: `http://twiki.di.uminho.pt/twiki/bin/view/Research/CROSS/WebHome`

its quality. It starts by gathering content in the package written in natural language, processing this content to compute metrics, and finally reasoning about these metrics to draw conclusions about the software quality.

Documentation analysis is also relevant in other research areas. Program Comprehension (PC) is an area of Software Engineering concerned with gathering information and provide knowledge about software to help programmers understand how a program works in order to ease software evolution and maintenance tasks [6]. Many of the techniques and methods used rely on mappings between program elements and the real world concepts these elements are addressing [7]. Non-source code content included in software packages can provide clues and valuable information to enhance the creation of these mappings. Program maintainers often rely on documentation to understand some key aspects of the software [9].

Assessing software quality for any given definition of quality is not easy [5] 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 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 addressees 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 described traits. Based on these conclusions a quantitative measure can be calculated about the quality of the non-source code content.

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 example of generated reports. Finally, Section 5 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) will be used for illustration purposes, mainly because it is small and produces outputs that can fit in the paper size without jeopardize reading.

[5] Available from `http://mama.indstate.edu/users/ice/tree/`

2 Related Work

Forward *et al* [2], in their survey about the general opinion of software professionals regarding the relevance of documentation and related tools, 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 [8], in his work about the requirements for open source software development, 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* [1] have identified documentation quality problems 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 focus on addressing this problem.

3 DMOSS Toolkit

The DMOSS toolkit main goal is to provide a set of tools that systematically process a software package and produce a final report with conclusions about the quality of the non-source content found in the package. This includes analyzing all the natural language text stored 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 tools that can be used by themselves, so that they can be useful in other contexts or environments. Higher order applications in the toolkit (applications that use the smaller tools) need to be modular so that new tools can be added without any 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.
– Look at the software package being analyzed as a tree. This allows the implementation of the analysis algorithms as a set of tree traversals. This keeps the implementation of the specific analysis algorithms self-contained.

Let us stress again the importance regarding the way DMOSS represents a software package: an annotated tree. In this tree, nodes represent files and

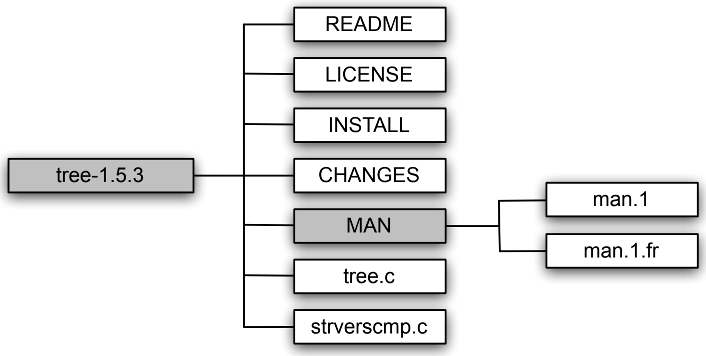

Fig. 1. Package tree like structure.

directories, and edges describe the hierarchical structure of the package. An example tree is illustrated in Fig. 1, for the `tree` software package.

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

1. During the first pass the goal is to gather informations about what files exist and what is their content. Each plugin is also run for each file individually, and the computed metrics are stored in the tree as node attributes;
2. In the second pass, results are aggregated. For each directory node the available metrics are reduced to a single result, and in the end, the root node (the package top directory) will contain the results of processing the entire package.

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 — and $Value$ 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:

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

where, $Path$ stores the file name and its path, $isFile$ is a boolean value stating if this node is a file or a directory, and $Text$ stores the natural language text found in the file. The $Text$ value is computed before starting the tree traversal stages.

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 [4], 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[6] the next step is to process the current file with all the available processors that support it, and store each processor result as a new node attribute.

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

Processors In order to compute attributes values for file nodes, the toolkit provides a 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).

New processors can be added or plugged in at anytime. Built in the toolkit there is also a table that states which file types can be handled by each processor. This helps to keep the traversing tree engine agnostic to which processors are available, and which files to process.

[6] The toolkit provides a set of plugins that implement several *processors*, and new plugins can be easily added.

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.
for $node \leftarrow tree : node.isFile == True$ do
    for $proc \leftarrow processors(node.type)$ do
        // Add processor resulting feature list to node
        $node.push(proc(node))$
return $tree$
```

Algorithm 2: Processor Example: Spell Checker

```
Input: node : Node
Result: New feature list to be added to the node.
$total \leftarrow 0$
$found \leftarrow 0$
for $word \leftarrow splitWords(node.text)$ do
    if $dictionary.valid(word)$ then
        $found++$
    $total++$
$f = Feature(attr = \text{"spellChecker"}, value = (total, found))$
return $[f]$
```

3.2 Second Pass: Reducing Results

The goal of the second tree traversal is to produce the final metrics results. This is achieved by combining 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 results for each sub-tree (and for each specific metrics). 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 filesystem `man/` directory) will have an attribute that stores the result of combining the spell checker processor result for files `man.1` and `man.1.fr`[7]. Later, this attribute value will be used to calculate the total result 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 These functions are used to reduce intermediate results, *i.e.*, combine the results found by the processors in the sub-tree of the node currently being

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

Algorithm 3: Reduce Package Tree

```
Input: tree : Tree representing package content.
Input: reducers : Set of available reducers.
Result: Tree after adding reducers results to nodes as features.
for $node \leftarrow tree : node \in Directories$ do
    for $reduce \leftarrow reducers$ do
        $node.push(reduce(node.children))$
return $tree$
```

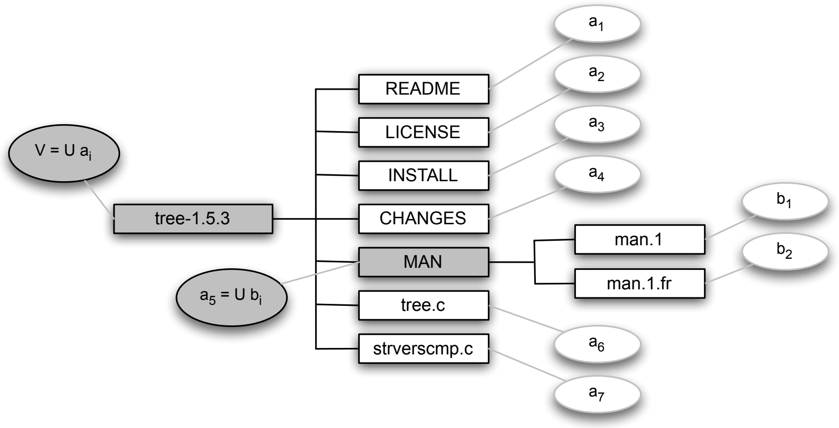

Fig. 2. Calculate final values recursively.

processed, and add this reduced result to the current node as a new attribute. 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 attributes per node, that confine 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[8] notation. An example of a HTML formatted report is illustrated in Figure 3.

In the end of the trees transversals 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 full tree is available as an associative array to be further processed by any other tool or application.

[8] http://www.graphviz.org/

Algorithm 4: Reducer Example: Spell Checker

```
Input: children : list of Nodes
Result: New set of features to be added to the node.
$totalAcc \leftarrow 0$
$foundAcc \leftarrow 0$
for $child \leftarrow children$ do
    $attr = child.getAttr(\text{"SpellChecker"})$
    $totalAcc \leftarrow totalAcc + fst(attr.value)$
    $foundAcc \leftarrow foundAcc + snd(attr.value)$
$new = Feature(name = \text{"spellChecker"}, value = (totalAcc, foundAcc))$
return $[new]$
```

Algorithm 5: Build Final Report

```
Input: tree : Tree representing package content.
Input: reporters : Set of available reporters.
Result: Final HTML report.
for $r \leftarrow reporters$ do
    $slice = sliceTree(r.features)$
    $final = final + r(slice)$
return $final$
```

Reporters The 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.

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 -file tree-1.5.3.tgz
Data saved as dmoss.data
```

This builds the tree representing the package, and executes all the tree transverses 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. 3, 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 *A*. 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*. Averaging all the features the final grade for the documentation in the package is *C*. 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. A final grade is given to the package (*C* in this report), which is the features' grade average.

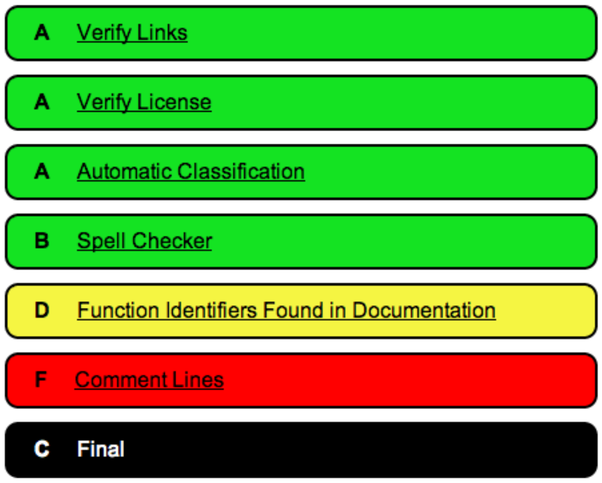
A Verify Links
A Verify License
A Automatic Classification
B Spell Checker
D Function Identifiers Found in Documentation
F Comment Lines
C Final

Fig. 3. Screenshot of a HTML report produced using DMOSS.

5 Conclusion

Non-source code content found in software packages provides useful insights and information about the application. This information can be used in many distinct areas: software certification, or source code understanding for software maintenance or evolution.

The DMOSS toolkit is able to process a software package as an attribute decorated tree, and has proven most valuable. Since once all the major engines (algorithms described in Section 3) were implemented, adding features to the analysis workflow is just a matter of adding a new plugin. This approach has allowed the development of a modular and pluggable toolkit, easy to maintain, and extend. The toolkit can process any package, regardless of programming language used, but the text extracting tool (from files) can require update for some specific archiving technologies.

Regarding the obtained results from software package analysis, we noticed that there is quite a lot of concern about overall package natural text information content. Nowadays, 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 less software engineering content. There are still some features that are more prone to have lower grades, for example the number of comment lines per lines of code.

Some tasks that can be performed 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.

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 - Fundação para a Ciência 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. J.C. Chen and S.J. Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992, 2009.
2. A. Forward and T.C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33. ACM, 2002.
3. Ø. Hauge, C. Ayala, and R. Conradi. Adoption of open source software in software-intensive organizations–a systematic literature review. *Information and Software Technology*, 52(11):1133–1154, 2010.
4. IANA. MIME Media Types. Web site: http://www.iana.org/assignments/media-types/index.html [Last accessed: 2012-11-27].
5. B. Kitchenham and S.L. Pfleeger. Software quality: the elusive target [special issues section]. *Software, IEEE*, 13(1):12–21, 1996.

6. Michael L. Nelson. A survey of reverse engineering and program comprehension. *CoRR*, abs/cs/0503068, 2005.
7. V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.
8. W. Scacchi. Understanding the requirements for developing open source software systems. In *Software, IEE Proceedings-*, volume 149, pages 24–39. IET, 2002.
9. B. Thomas and S. Tilley. Documentation for software engineers: what is needed to aid system understanding? In *Proceedings of the 19th annual international conference on Computer documentation*, pages 235–236. ACM, 2001.